TAMPERE UNIVERSITY OF TECHNOLOGY

ANURADHA DANDE

SIMULATION OF MULTIPROCESSOR SYSTEM SCHEDULING

Master of Science Thesis

Examiner(s): Professor Jari Nurmi,
Doctor Sanna Määttä
Examiner and topic approved by the
Teaching and research Council on
06 April 2011

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
ANURADHA DANDE: Simulation of Multiprocessor System Scheduling
Master of Science Thesis, 48 pages + 20 Appendix pages
February 2014
Major: Software Systems
Examiner(s): Professor Jari Nurmi, Doctor Sanna Määttä
Keywords: Multiprocessor system, Scheduling, Simulation, Efficiency of the system.

The speed and performance of computers have become a major concern today. Multiprocessor systems are introduced to share the work load between the processors. By sharing the work load among processors, the speed and efficiency of the system can be increased drastically. To share the workload properly between the processors in the multiprocessor system a proper scheduling algorithm is needed. Hence, one of the major factors that influence the speed and efficiency of the multiprocessor system is scheduling. In this thesis, the main focus is on the process scheduling for multiprocessor systems. The factors which influence scheduling and scheduling algorithms are discussed.

Based on this idea of sharing the load among processors in the multiprocessor system, a simulation model for scheduling in a symmetric multiprocessor system is developed in the Department of Digital and Computer systems at Tampere University of Technology. This model treats all the processors in the system equally by allocating equal processor time for all the processes in the task and also evaluates the total execution time of the system for processing an input job. The scheduling algorithm in this simulation model is based on the input processes priority. The necessity of scheduling in multiprocessor systems is elaborated.

The goal of this thesis is to analyse how process scheduling influences the speed of the multiprocessor system. Also, the difference in total execution time of the input jobs with different number of processors and capacity of the processors in the multiprocessor system is studied.

PREFACE

The research work in this thesis was done at department of Digital and computer systems in Tampere University of technology. The aim of this project is to analyze how scheduling influences the speed of multiprocessor system. A simulation model of multiprocessor system scheduling is also implemented with the thesis.

I would like to thank Professor Jari Nurmi for his support. I especially thank Doctor Sanna Määttä for her continuous guidance and help in writing this thesis. I also thank Jarno Mannelin for his help in QT related issues. I sincerely admit that without the help of the above mentioned people the completion of this work would not have possible.

Finally I thank my loving husband Chandra Sekhar and daughter Rishitha for their support and cooperation while writing the thesis.

Tampere, 18.03.2014

Anuradha Dande

TABLE OF CONTENTS

Abstract i
Preface ii
Table of Contents iii
Terms and Definitions v
1. Introduction 1
2. History of Multiprocessor Systems 4
    2.1 Amdahl's Law 5
    2.2 Moore's Law 6
3. Architecture of Multiprocessor System 8
    3.1 Processor Symmetry 8
        3.1.1 Symmetric Multiprocessing 8
        3.1.2 Asymmetric Multiprocessing 9
        3.1.3 Symmetric verses Asymmetric Multiprocessing 10
    3.2 Memory Access 11
        3.2.1 Uniform Memory Access Systems 11
        3.2.2 Non-Uniform Memory Access Systems 11
        3.2.3 Massively Parallel Systems 12
    3.3 Processor Coupling 13
        3.3.1 Tightly-coupled Multiprocessor Systems 13
        3.3.2 Loosely-coupled Multiprocessor Systems 13
        3.3.3 Tightly-coupled verses Loosely-coupled Multiprocessor Systems 14
    3.4 Software Implementation in Multiprocessor Systems 14
        3.4.1 Single Instruction Single Data Processing 15
        3.4.2 Single Instruction Multiple Data Processing 15
        3.4.3 Multiple Instruction Single Data Processing 16
        3.4.4 Multiple Instruction Multiple Data Processing 17
    3.5 Issues in Multiprocessor Systems 18
4. Concurrency in Multiprocessor Systems 20
    4.1 Programming Models 20
        4.1.1 Independent Parallelism 20
        4.1.2 Regular Parallelism 21
        4.1.3 Unstructured Parallelism 21
    4.2 Concurrency control 21
        4.2.1 Critical section 21
        4.2.2 Mutex Lock 22
        4.2.3 Semaphore 22
    4.3 Concurrency implementation 22
5. Scheduling in Multiprocessor System 24
    5.1 Scheduling Algorithms 24

5.1.1 First In First Out Scheduling Algorithm 25
5.1.2 Shortest Job First Scheduling Algorithm 25
5.1.3 Priority Based Scheduling Algorithm 25
5.1.4 Round Robin Scheduling Algorithm 26
5.1.5 Multilevel Queue Scheduling Algorithm 26
5.2 Multiprocessor System Scheduling 27
5.2.1 Scheduling Criteria 27
5.2.2 Implementation of Scheduling in Multiprocessor Systems 28
5.2.3 Choosing a Scheduling Algorithm 29
5.2.4 Problems in Multiprocessor Scheduling 30
6. Simulation in Multiprocessor Systems 31
7. Design and Implementation of Simulation Model for Multiprocessor System Scheduling 33
7.1 Scheduling Criteria in the Simulation Model 33
7.2 Concept Model 34
7.3 Design of Scheduler 35
7.4 Example 37
7.5 Technologies Used 39
7.6 Graphical User Interface, Inputs and Outputs 39
7.6.1 Graphical User Interface 39
7.6.2 Inputs 40
7.6.3 Outputs 40
7.7 Result Analysis 41
7.7.1 Analysis 1 41
7.7.2 Analysis 2 42
8. Conclusion and Future work 44
References 45
APPENDIXES 49

TERMS AND DEFINITIONS

Task	Unit of execution in a job given to the system for execution.
Process	Sub part of a task which is executed. Tasks contain several processes.
Processor	Customization of the system to meet specific needs of the end user. The system is customized by configuring the system parameters.
Concurrency	Two or more processes executing in parallel
Scheduling	Way of allocating processor time to processes for execution
SMP	Symmetric Multiprocessors
ASMP	Asymmetric Multiprocessors
UMA	Uniform Memory Access
NUMA	Non Uniform Memory Access
MPP	Massively Parallel Multiprocessors
GUI	Graphical User Interface

1. INTRODUCTION

Speed and performance of the computer systems are the major concerns in the computer industry. To increase the speed and performance of the system, the total execution time of the input job should be reduced. The traditional ways to reduce the total execution time of the input job in the system is by

- Increasing the transmission speed in the processor
- Increasing computational power of the processor [16]

To increase the transmission speed inside the processor, the high speed modules in the processor must be placed very close to one another. The transmission speed will be increased with the decrease in distance between the high speed modules to an extent, but after a limit, this will not affect the speed the system much. [16]

The computational power of processor depends on number of transistors in the processor. Right now, processors contain millions of transistors. Increasing the number of transistors on processors is a very difficult and expensive. Hence increasing computational power of processor also has limitation. [16]

One of the best and efficient ways to increase the speed and performance of the system is through parallel processing. Parallel processing is executing different parts of the same job simultaneously. This reduces the total execution time of the input job. Multiprocessor systems contain more than one processor, through which parallel processing is possible. All these processors in the multiprocessor system share the work load and execute in parallel. [16]

The work load or the input job contains one or more tasks and each task contains one or more processes. These processes are shared among the processors in the multiprocessor system. By distributing the work load among the processors in the system, the total execution time will be decreased drastically. [1]

Multiprocessor systems are very complicated as they are capable of parallel execution. To perform all the processes in the task properly, they must be efficiently executed. For an efficient execution, all the processes must be sched-

uled properly. Scheduling is the way of allocating processor time and resources to the processes for execution. [1]

Distribution of the processes among the processors in the multiprocessor system is done based on scheduling criteria of the system. The scheduling criteria depend on the goals of scheduling and hardware architecture of the system. These scheduling criteria vary from system to system [3]. The common goals of scheduling are

- Minimizing the total execution time of the system
- Increasing the throughput
- Minimizing the waiting time of the processes
- Maximizing the resource utilization
- Sharing processor time for each process equally
- Minimizing the communication delay
- Handling task prioritization [1]

As the efficiency of scheduling reflects the speed of execution in the system, multiprocessor system scheduling has become one of the most studied problems today. [3]

Scheduling is implemented by using appropriate scheduling algorithm. A scheduling algorithm is selected and implemented according to the scheduling criteria of the multiprocessor system. The order of execution of the process is decided by the scheduling algorithm. [1]

In general, tasks are executed more quickly in multiprocessor system than when run on a uniprocessor system. For smooth execution, communication between the processors in the system is necessary. Also, processes in the system need to share memory and resources. In cases where the estimated execution time of the task is small and if the communication between the processes, use of shared resources is very frequent then uniprocessor systems may take advantage over multiprocessing systems in terms of execution time. [1]

Multiprocessor systems are very complex compared to uniprocessor systems. Handling problems such as communication delays, resource allocations, error debugging in multiprocessor systems is difficult [16]. As the multiprocessor systems are complex, simulation of multiprocessor systems helps to understand and learn the internal functionality easily. [1]

As a part of the thesis work, a simulation model of multiprocessor system scheduling is developed using SystemC and QT. The basic aim of this simulation model is

- To understand the change in execution speed with the change in number of processors and efficiency of processors in the multiprocessor system.
- To learn how priority of processes affects the order of execution of the processes.
- To handle pre-emption of the processes while execution.

The rest of the thesis is organized as follows. Chapter 2 outlines the evolution of multiprocessing technology over years. Chapter 3 discusses the architecture of multiprocessing systems. Chapter 4 gives a brief idea of concurrency. Chapter 5 gives an overall idea of scheduling in multiprocessing system. In Chapter 6, simulation in multiprocessor system is explained. Chapter 7 discusses about design and implementation of simulation model of multiprocessor system. Conclusion and future work are in chapter 8.

2. HISTORY OF MULTIPROCESSOR SYSTEMS

The performance of a system can be increased by increasing the performance of the processor. The performance of the processor can be increased by increasing the clock rate [4]. During the evolution of computers, the size of processors was reduced with the decrease in size of transistors. With decrease in size of transistors, more transistors were placed in the processors, thus allowing them to operate with very high clock rates and high computational power [16]. The increase in the number of transistors resulting in increase in speed of processors over period of time is explained by Moore's law, which was proposed by Gordon E. Moore in 1970 [27]. Moore's law is briefly explained in section 2.2.

The clock rate of the processor cannot be increased above a certain level due to physical limitations. In that case, the system performance can be increased further by increasing the number of processors in the system which can share the workload and execute in parallel. [4]

The technique of sharing the work load among two or more processors in the system and allowing these processors to work in parallel is called multiprocessing. The systems which carry out multiprocessing are called multiprocessor systems [1]. The concept of multiprocessing created much of interest among companies and users in many fields. Depending on technology and market need, a number of companies started developing multiprocessor systems by mid 1950s. [4]

In 1960's, parallel processing systems with a cross switch that connects the processors with input-output devices to memory are designed, these systems are called mainframe systems. These systems are capable of supporting hundreds and thousands of users simultaneously. Mainframe systems found extensive use in public governing systems such as universities and hospitals. Super computers were introduced in early 1980's, with high throughput and high performance using parallel processing. These super computers are faster and capable of doing complex tasks such as predicting weather forecast. In 1970's, parallel processing mini computers are introduced for using in large organizations such as hospitals which can support many users simultaneously. In 1980's, microcomputers became popular as they were used as standalone per-

sonal computers. These systems are capable of parallel processing and are used for personal computers and in offices. [24] [30]

The increase in speed of the overall system by improving a part of system is explained by Amdahl's Law. By using Amdahl's Law, the change in the speed of multiprocessor system with the increase in number of processors is briefly explained in section 2.1. [29]

2.1 Amdahl's Law

Amdahl's law states that the performance of the system can be increased by increasing the performance of a portion of the system. This theory can be applied to the multiprocessor systems to calculate the theoretical performance of the system. [4]

$$Speedup = \frac{1}{F + (1 - F) / N}$$

Figure 2.1 Amdahl's equation for multiprocessing [4]

In Figure 2.1, N is the number of processors in the system and F is the portion of the system which is sequential and cannot be run in parallel. This Amdahl's equation (in Figure 2.1) is used to calculate the maximum performance improvement of a multiprocessing system theoretically. The theoretical calculation of the performance improvement in the system with the increase in number of processors by using Amdahl's law is shown in Figure 2.2. [4]

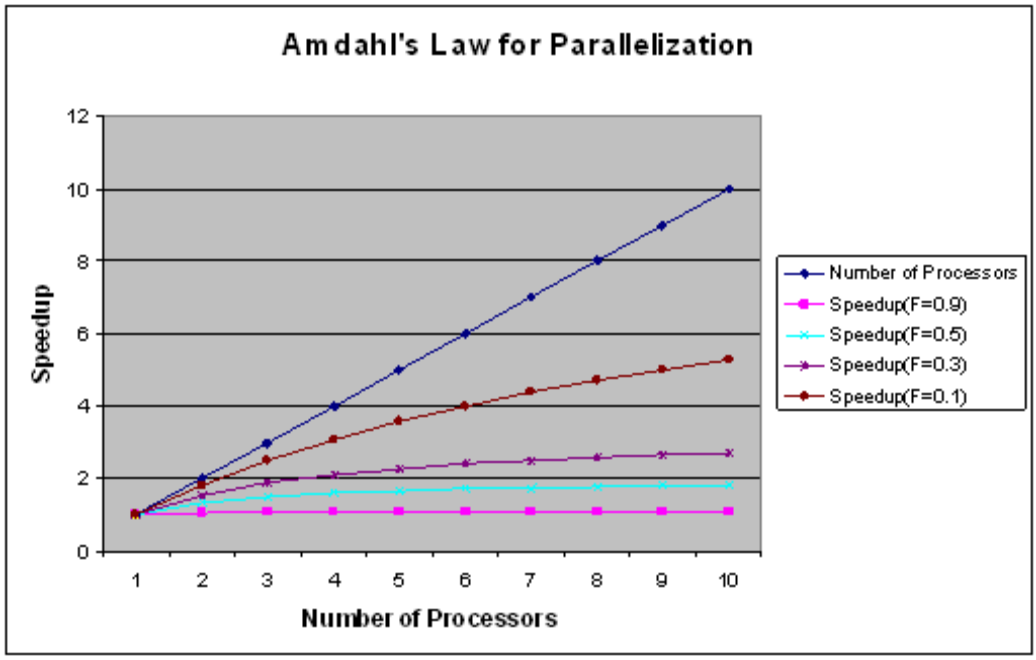

Figure 2.2 Amdahl's law for up to ten CPUs [4]

In Figure 2.2, the blue line shows the number of processors. If the F factor (the portion of the system which is sequential and cannot be run in parallel) is increased, the speed of the system is reduced, even after increasing the number of processors. This can be noticed from the purple and the brown lines in the Figure 2.2 [4]

2.2 Moore's Law

Moore's law states that" the number of transistors incorporated in a chip will approximately double every 24 months." In spite of existing power consumption issues, Moore's law is still applicable today [28]. The graph in the Figure 2.3 illustrates Moore's law for Intel chips over a time period from 1970-2005. [32]

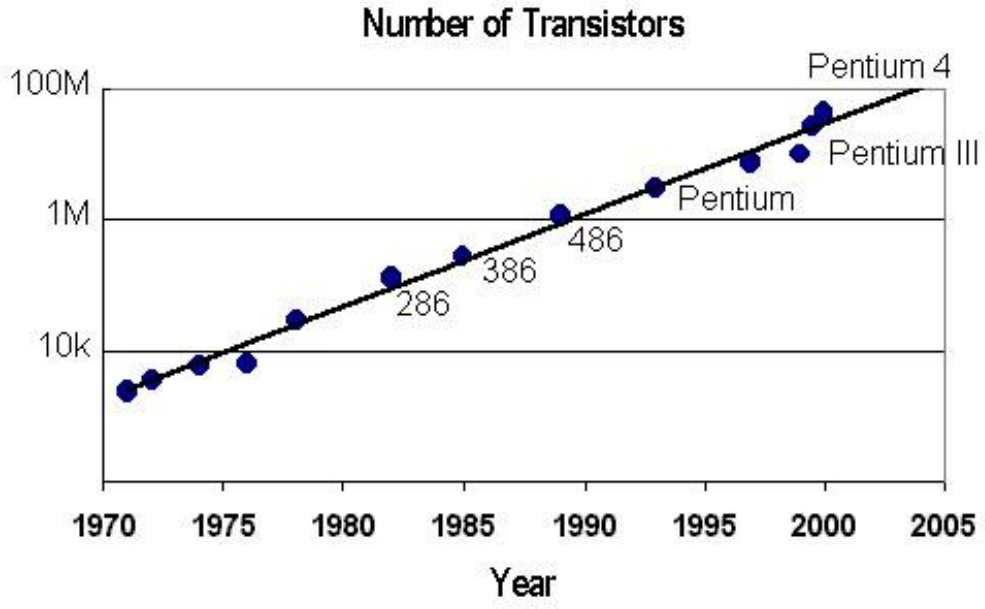

Figure 2.3 Increase in number of transistors in Intel chips over years[32]

The Figure 2.3 shows the transistor count in Intel chips from 1970's to 2005. From the straight line in the Figure 2.3, it is understood that the number of transistors is almost doubled in every 2 years in Intel chips. [32]

The system performance also increased in time by utilizing all the processors efficiently using parallel processing. This is depicted in Figure 2.4 for Intel multiprocessor systems. [31]

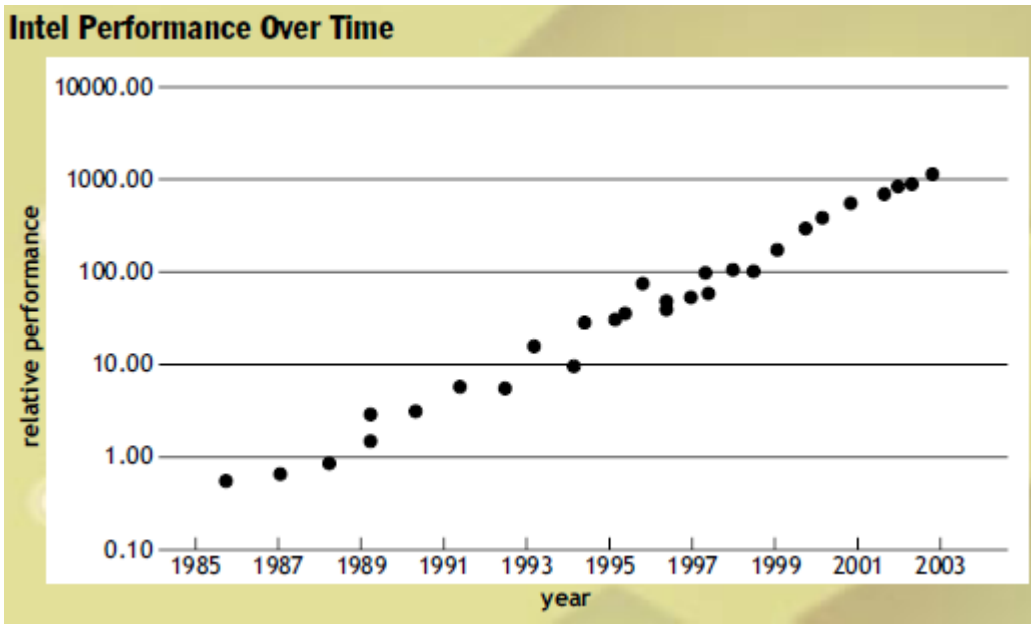

Figure 2.4 Intel performance over time [31]

Figure 2.4 shows constant growth in relative performance of Intel multiprocessor systems from 1980 to 2003.

3. ARCHITECTURE OF MULTIPROCESSOR SYSTEM

Compared to uniprocessor systems, multiprocessing systems are capable of executing multiple processes in parallel by sharing the work load among all the processors in the system. The way of execution of the processes in the multiprocessing systems depends on the architecture of the system [1]. This section explains different kinds of hardware and software architectures of multiprocessor systems. There are different kinds of multiprocessing systems which are classified based on their symmetry, coupling, memory access and software implementation. This chapter gives brief idea of classification of multiprocessor systems.

Multiprocessor hardware implementations are carried out based on the way how all the processors are connected in a multiprocessing system, how the resources are shared among the processors and how communication between the processors is made. Multiprocessor software implementations are carried out based on data use and instructions processed by processors. [5][1]

3.1 Processor Symmetry

Symmetry of a system is defined by both the software and hardware design constraints. For example, a multiprocessor system may utilize all its processors equally or it may reserve few processors for special tasks such as responding to priority interrupts. The multiprocessor systems are designed based on the kind of jobs executed by the system. The design takes care of utilizing all the processors in the system. Based on the processor symmetry, multiprocessing can be classified as symmetric and asymmetric multiprocessing. [5]

3.1.1 Symmetric Multiprocessing

In symmetric multiprocessing (SMP) systems, all the processors are treated equally. Here all the processors are connected in such a way that they share all the tasks and resources equally. Symmetric multiprocessing architecture is widely used. [5]

As all the processors are capable of working equally, the symmetric multiprocessing architecture balances the work load on the system efficiently by transferring and sharing the task equally among all the processors in the system. Figure 3.1 shows a typical symmetric multiprocessor system where all the processors are connected to the same memory through a bus. In symmetric multiprocessing, all the processors run on single operating system allowing parallel processing. [5]

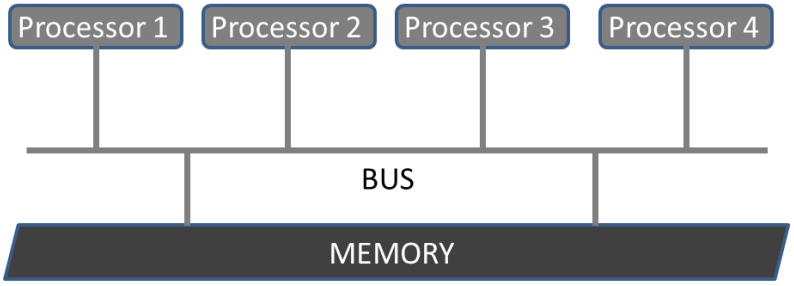

Figure 3.1 Symmetric multiprocessor system [5]

3.1.2 Asymmetric Multiprocessing

Unlike symmetric multiprocessing architecture, in asymmetric multiprocessing architecture processors are not treated equally. As few processors will be reserved for executing only certain tasks such as handling interrupts, the entire work load is not shared equally between all the processors. Hence the workload between the processors varies, as the work load cannot be moved among all the processors equally. This kind of multiprocessing system is designed for executing specific tasks. [20] [1]

Asymmetric multiprocessing architecture is complex and difficult to implement and understand. The processors in asymmetric multiprocessing architecture may have different operating systems. The processors in this architecture need not be aware of each another. [20]

Asymmetric multiprocessing system architecture uses master-slave mechanism. The master processor verifies the priority and kind of the task, divides the workload accordingly and assigns the task to the appropriate slave processors to finish the task. Slave processors just execute the tasks given by the master processor. Due to this, work load may not be shared equally between the processors. Few processors may also be idle in this kind of execution. [5] [1]

Figure 3.2 and Figure 3.3 show examples of asymmetric multiprocessor system. In Figure 3.2, all the processors do not have same access to the resources of the system. Processor 1 and processor 3 have their private memory and all the processors have access to memory and Input/Output devices. In the asymmetric multiprocessor system in Figure 3.3, only processor 4 has access to Input and Output devices. [5]

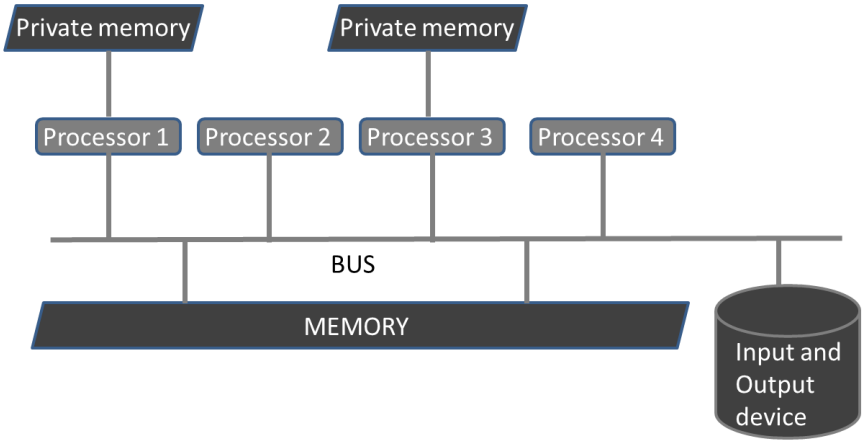

Figure 3.2 Example1 of Asymmetric multiprocessor system [5]

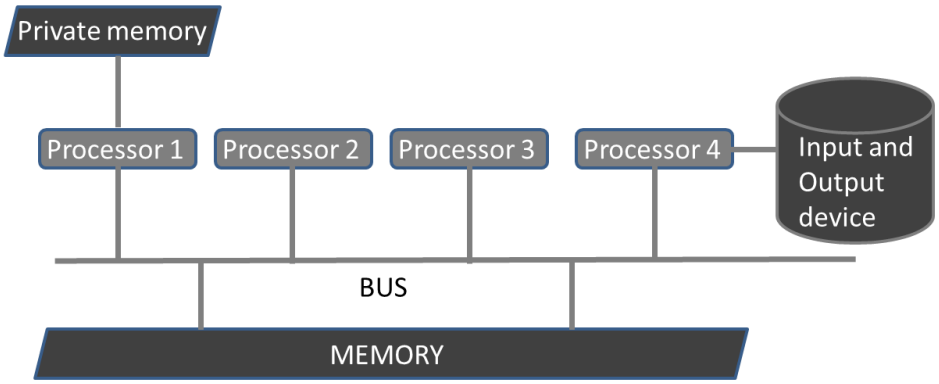

Figure 3.3 Example2 of Asymmetric multiprocessor system [5]

3.1.3 Symmetric verses Asymmetric Multiprocessing

All the processors in symmetric multiprocessing systems are treated equally and all of them are capable of executing a given task. As asymmetric multiprocessing systems work on master-slave architecture all the processors are not capable of executing all kinds of tasks. Master processor decides which processor executes which task. [20]

Implementing asymmetric multiprocessor architecture is very difficult compared to symmetric multiprocessing architecture. A programmer should take special care while designing asymmetric multiprocessing architecture. [20]

3.2 Memory Access

Based on sharing of memory, multiprocessors can be categorised as [35]

1. Shared memory systems
2. Shared disk systems
3. No sharing systems.

Based on memory access of the processors, multiprocessor systems are classified as Uniform Memory Access systems (UMA) and Non-Uniform Memory Access systems (NUMA). [35]

3.2.1 Uniform Memory Access Systems

All the processors in the multiprocessor system have uniform access to the memory in the system. Figure 3.4, illustrates uniform memory access systems model. [35]

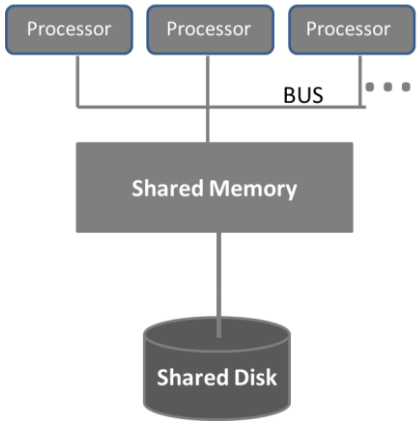

Figure 3.4 Uniform memory access systems (UMA) [35]

3.2.2 Non-Uniform Memory Access Systems

All the processors in the multiprocessor system will not have uniform access to the memory in the system. This can be seen in the Figure 3.5 Non-Uniform memory access systems model. [35]

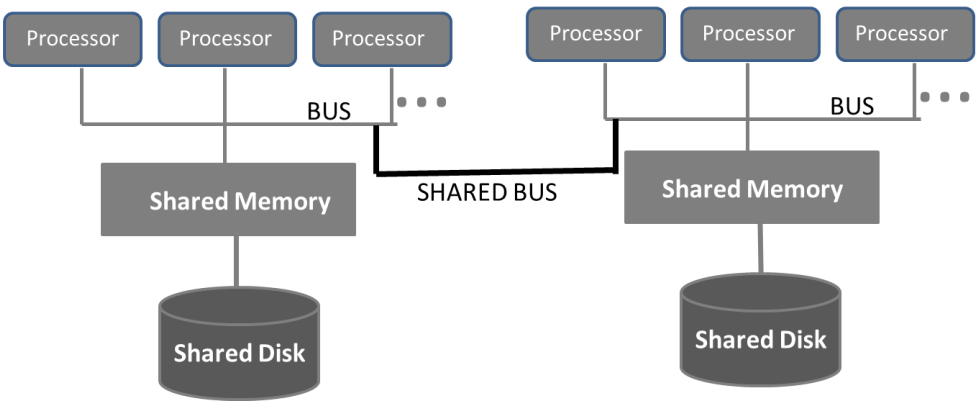

Figure 3.5 Non-Uniform memory access systems (NUMA) [35]

3.2.3 Massively Parallel Systems

Massively parallel systems do not share a common memory, but they have access to common disks. These systems may vary from few nodes to thousands of nodes. All the nodes will be arranged in grid, mesh or hyper cube model. Every node will have its own private memory and devices. Most of these massively parallel systems are designed in a way that on failure of a node, other systems can access failed system resources. Figure 3.6 shows massively parallel systems model. [35]

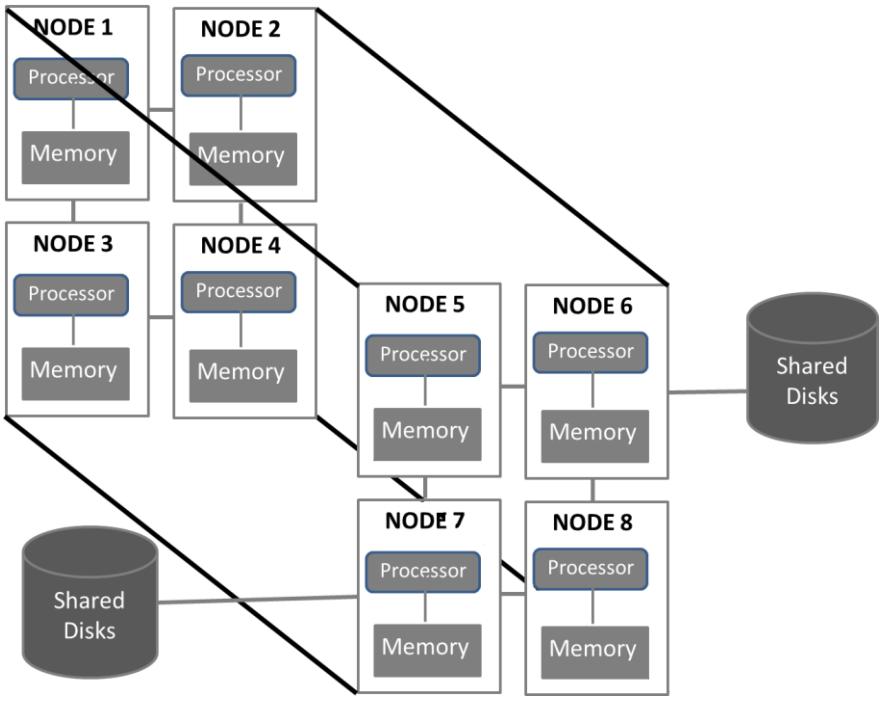

Figure 3.6 Massively parallel systems model. [35]

3.3 Processor Coupling

Based on the way of connectivity and communication in the system, processor coupling is classified in to tightly-coupled multiprocessor systems and loosely-coupled multiprocessor systems.

3.3.1 Tightly-coupled Multiprocessor Systems

All the processors in a tightly-coupled multiprocessing system are connected via bus and all the resources are shared equally among all the processors in the system. Communication between the processors is carried out with the help of shared memory in the system. Bandwidth of the bus, memory storage, retrieval, and message passing techniques determine the performance in these systems. Figure 3.7 shows an example of tightly-coupled multiprocessor system. [5][35]

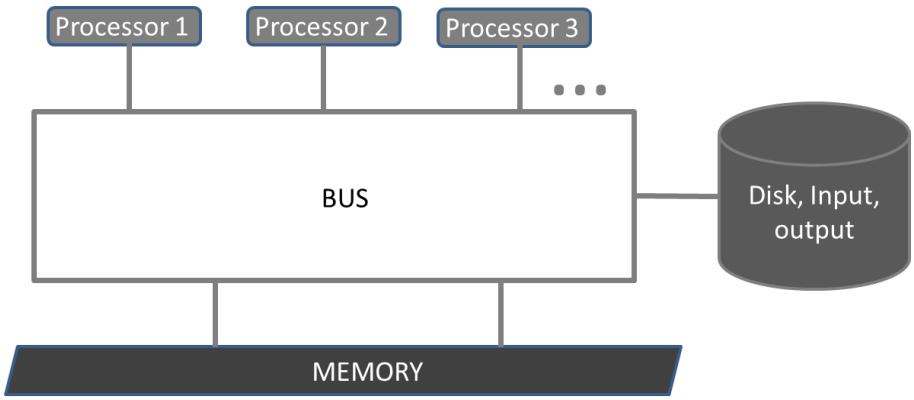

Figure 3.7 Tightly-coupled multiprocessor system [34][35]

This kind of coupling is used in chip multiprocessing or multi-core computing. All the processors on a single chip are tightly-coupled. Multiprocessor mainframe systems are generally tightly-coupled. [5]

3.3.2 Loosely-coupled Multiprocessor Systems

There is no regular pattern in the connectivity of processors in loosely-coupled multiprocessor systems. All the processors are treated as clusters. Mostly these types of systems are connected with high speed Ethernet. The Figure 3.8 shows an example of loosely-coupled multiprocessor system. Node 1, Node 2, Node 3 are clusters connected to common disks via bus. [5]

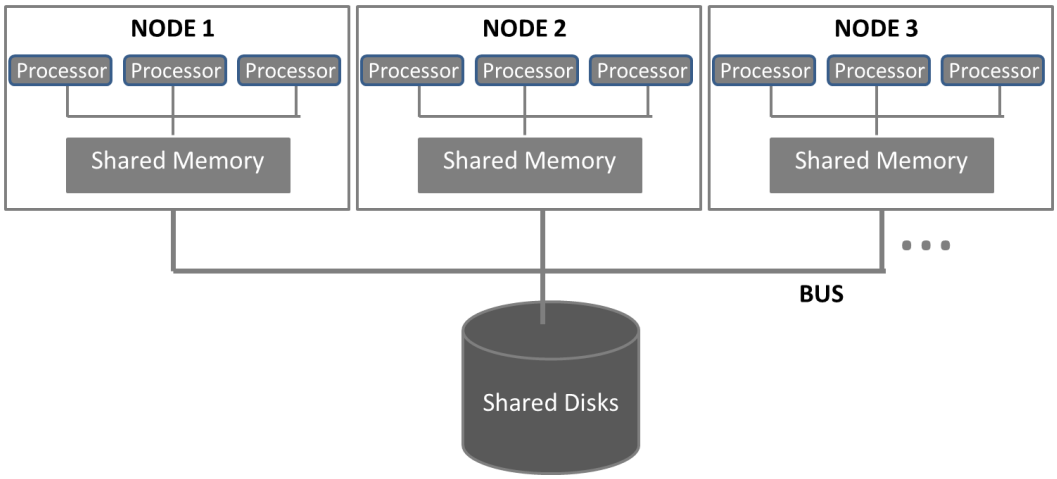

Figure 3.8 Loosely-coupled multiprocessor system [35]

3.3.3 Tightly-coupled verses Loosely-coupled Multiprocessor Systems

Compared to loosely-coupled multiprocessor systems, tightly coupled multiprocessor systems give better performance. Tightly coupled multiprocessor systems occupy smaller area than loosely coupled multiprocessor systems. During the design of tightly coupled systems, by allowing relevant components to work together, they can be made more efficient in terms of energy consumption. [5]

The initial costs incurred in establishing tightly coupled multiprocessing systems are higher than loosely coupled multiprocessor systems. When detached from cluster, loosely coupled machines can be used as independent machines hence it is easier to recycle loosely coupled systems. [5]

3.4 Software Implementation in Multiprocessor Systems

Based on data use and instructions processed by processors in computer systems, software implementations in computer systems are classified. Figure 3.9 shows the same. Flynn's taxonomy explains this classification in detail. [21]

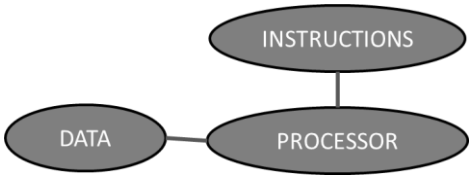

Figure 3.9 Inputs for Flynn's taxonomy [36]

Different combinations of input data and instructions given to the systems are discussed in the Flynn's taxonomy. Table 3.10 shows the same. Based on the-

se combinations of input data and instructions, software architecture of the system is classified in to 4 categories: [5] [21]

1. Single Instruction Single Data processing (SISD)
2. Single Instruction Multiple Data processing (SIMD)
3. Multiple Instruction Single Data processing (MISD)
4. Multiple Instruction Multiple Data processing (MIMD)

	Single Instruction	**Multiple Instruction**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Table 3.10 Flynn's taxonomy [21]

Parallel processing is carried out by SIMD processing and MIMD processing. [21]

3.4.1 Single Instruction Single Data Processing

In SISD processing, each processor processes only single instruction by accessing data from a single memory source. Here, each processor sequentially processes instructions (pipelined execution) where each instruction can access one data item. Figure 3.11 represents the same. [5]

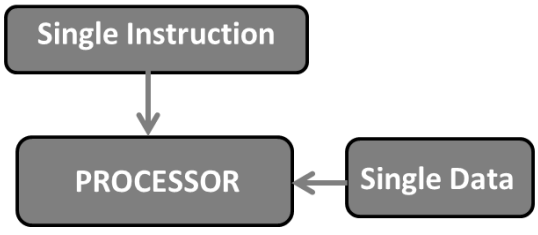

Figure 3.11 Single Instruction Single Data processing [36]

3.4.2 Single Instruction Multiple Data Processing

In a SIMD processing, the instructions are processed one after other in a sequence by accessing data from multiple data sources in parallel. [5]

In SIMD processing, large chunks of data is divided in to sub parts so that similar and independent operations can be operated in parallel. Figure 3.12 explains the same. A set of instructions are executed by the processors in the system one by one, where each processor performs similar operation on the sub parts of chunk of data from multiple resources. This kind of single instruction multiple data processing is used in parallel or vector processing systems where similar operations are made on large chunks of data. This architecture reduces the elapsed time in completing a given task. Thus increases the performance of the system. [5]

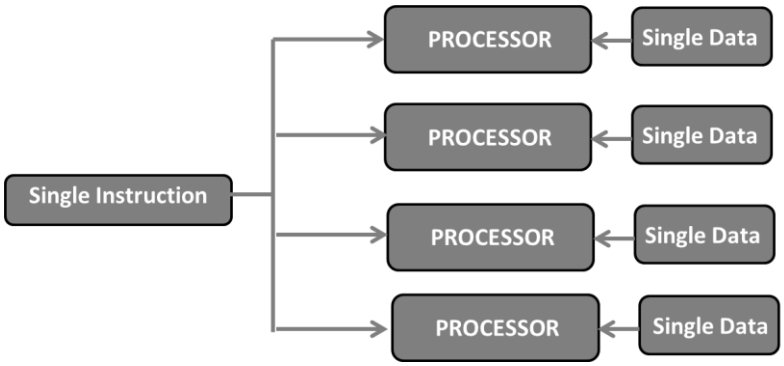

Figure 3.12 Single instruction multiple data processing. [36]

In this architecture, proper care should be taken in dividing the data to several sub parts. Special optimizing compilers are used in this SIMD processing systems to divide the data to several sub parts. This kind of architectures is used for computer simulations. [5]

3.4.3 Multiple Instruction Single Data Processing

MISD processing is similar to parallel computing architecture where multiple instructions are executed by different processors on same data. Figure 3.13 shows the same. These systems are also fault tolerant. There is no chance of getting wrong results even if one of the processor in the system fails abruptly. These systems are very expensive and do not improve performance. [5]

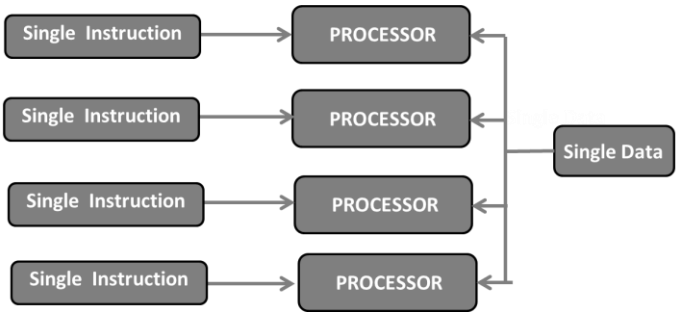

Figure 3.13 Multiple instruction single data processing [36]

3.4.4 Multiple Instruction Multiple Data Processing

In MIMD architecture multiple processors in the system can execute multiple instructions on multiple data. This architecture is used for those systems where instructions in the tasks can be executed independently in parallel. Figure 3.14 explains the same. Shared or distributed memory is used in this architecture system. Using this architecture, parallel processing is achieved. Distributed systems use this architecture. [5][21]

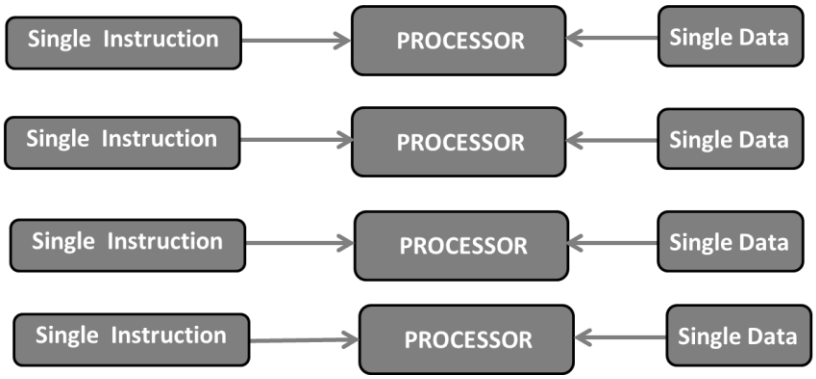

Figure 3.14 Multiple instruction multiple data processing [36]

This architecture is used in the areas such as computer-aided design, computer-aided manufacturing, simulation, modelling and communication switches. Implementing this architecture is easy but the issues such as races (more than one processes requesting for same resource), deadlocks and resource utilization are common. To avoid or handle such issues critical sections, locks and semaphores are used. [5]

3.5 Issues in Multiprocessor Systems

This chapter discusses the hardware and software issues related to multiprocessor systems. Multiprocessing systems are very complex thus understanding the system is also difficult. For implementing multiprocessing in the system, the input job must be divided and shared by all the processors in the system. Dividing the entire input job to smaller tasks and processes is not easy. Special software should be available in the multiprocessor systems to transform the main job in to dividable form, so that splitting the input job to smaller tasks is done meaningfully and efficiently. Scheduling is needed to allocate the processes among the processors in the system. Balancing the work load between all the processors is also a very challenging issue. [14]

Without proper synchronization among the processors in the system, data accessing is not reliable and may lead to wrong results. Tracking the execution of events in the multiprocessor systems is not easy [14]. Due to unexpected latencies (time delay between request and response) in the system, establishing communication between processors and accessing resources is hard. [15]

In multiprocessor systems, synchronization issues may also occur if two or more processes needs access for a shared resource at the same time, this results in race situation. Races are very common in multiprocessing environment. Races are very harmful and difficult to resolve. Predicting race conditions while scheduling is also difficult. As avoiding races will result to serialization, races must be resolved very carefully. Races are resolved using critical sections, semaphores and mutex locks. [1]

Critical section is the section in which only one process is allowed to access shared resources and no other process can enter their critical sections. To synchronize different processes entering critical section to access shared resources, mutex locks or Semaphores are used. Mutex locks provide mutual exclusion. Mutex locks control the processes entering critical section by locking the process, while entering in critical section and releasing the locks while exiting critical section. Mutex locks help to prevent race conditions. Deadlocks may occur in situations when two or more processes need two or more shared resources in order to finish the task and if they are waiting indefinitely for the another to release the acquired locks of a shared resource. Deadlock detecting and resolving is complex and expensive. [1]

Testing the multiprocessor systems is difficult. This is because tracking the execution of events in the multiprocessor systems is not easy. [41]

Issues such as network delays or power consumption are very crucial in multiprocessor systems. As the devices are manufactured smaller day by day, millions of transistors are packed in a single chip, which consumes more power and emits a lot of heat. To control this heat, additional hardware is needed for cooling and handling power consumptions in multiprocessor systems. [14] [3]

4. CONCURRENCY IN MULTIPROCESSOR SYSTEMS

Concurrency is a property where several processes in a task execute in parallel by interacting with each other to complete the task. Concurrent computing programming models are designed to implement concurrency. The concurrent computation is carried in multiprocessor systems. [1]

4.1 Programming Models

A programming model defines the method of execution of the tasks. In general, programming models are designed by the kind of problems handled and the kind of system architecture where the model is deployed. A parallel programming model defines the method of execution of parallel tasks under different circumstances. These parallel programming models generally depend on the way the job is divided in to sub tasks to execute in parallel and kind of interactions or dependencies between the tasks. [9]

Problems related in implementation of concurrency are not same always. Based on the problem and type of solution, a specific programming model is designed. Hence, various programming models evolved during time. [9]

4.1.1 Independent Parallelism

In independent parallelism all the tasks are executed independently and in parallel. Independent parallelism is possible only when there is absolutely no dependency or no interactions between the computations in the task. [9] [10]

For example, consider an application that needs to sort two data sets and merge the two data sets finally. As there is no dependency between the two data sets they can be sorted independently and in parallel. Finally, they can be merged. [10]

4.1.2 Regular Parallelism

Regular parallelism deals with executing mutually dependent tasks in parallel. Regular parallel programs require synchronization of computations and careful monitoring of the computations in the task. In this kind of parallel programs, data sharing and mutual dependencies should be analysed thoroughly to determine how to execute them in parallel. However, analysing the programs in advance is not always possible as it is impossible for compilers to restructure and understand all kinds of complex programs at all times. [9]

For example, if there are several tasks in an application sharing the same resource, then to maximize parallel execution of the tasks in the application all the computations should be carefully monitored and the execution of the tasks should be synchronized properly so that there is no overlapping in accessing the resource [9]

4.1.3 Unstructured Parallelism

Unstructured parallelism arises in the situation when the tasks in the application are least disciplined and nondeterministic. In this kind of applications all the tasks should be coordinated explicitly. Otherwise the execution of the application will be nondeterministic. [9]

For example, in complex applications with common memory where accessing data by the tasks is not predictable, the conflicting data access tasks execution should be explicitly monitored and synchronized. [9]

4.2 Concurrency control

Concurrency control is necessary to avoid ambiguity among concurrently executing process in processing requests for allocating same shared resources at the same time. To achieve concurrency control and synchronize processes, mechanisms such as critical sections, mutex locks and semaphores are used. [1]

4.2.1 Critical section

Races among the processes will arise when two or more processes try to acquire access for same shared resource at the same time. To avoid such race situations and to ensure smooth concurrent execution, critical sections are used. [1]

Critical section is a segment in which only one process can enter and access the shared resources. When one process is in its critical section for accessing the resources, no other parallel executing process can enter in its critical section. Critical section must provide mutual exclusion, which means at the same time, no two processes can execute in their critical section. Processes using the critical section are monitored by entry and exit status. Critical sections control concurrency by synchronizing the processes in execution. [1]

4.2.2 Mutex Lock

To control processes in accessing critical section and prevent race conditions mutex (mutual exclusion) locks are needed. Using mutex locks, processes are locked while entering critical section and released while exiting the critical section. Basic use of mutex locks is to provide mutual exclusion. Mutex locks are implemented using boolean variables, to indicate the availability of the critical section. Other processes if need to access the critical section, then stays in busy waiting state for their turn to acquire the critical section. The processes in busy waiting state checks continuously at regular intervals for the availability of the critical section. One of the disadvantages of busy waiting is that during wait the processor time (cycles) is consumed. [1]

4.2.3 Semaphore

Semaphores use signals for giving access for processes to critical section. Semaphores control the access to predefined number of instances of critical sections by incrementing and decrementing the values of the semaphore variable. [1]

4.3 Concurrency implementation

The first step in concurrency implementation is identifying the processes in the task that can be executed in parallel. After identifying all the processes that can be parallelized, they should be analysed based on their granularity, order of execution, resources needed to execute and dependency on other processes. Based on this analysis, the parallelizable processes should be scheduled for execution. Accordingly, a scheduling algorithm to execute the task is designed. [13]

The understanding of the tasks and the internal processes dependencies affect the algorithm design of parallel programming. To ensure determinism in the ex-

ecution, scheduling of the processes should be controlled using external algorithms than by system. Detailed study of the system and the input jobs is necessary while designing algorithm for communication and synchronization mechanisms between the processes and shared resources. [13]

After the algorithm design and implementation for parallel execution of processes, program optimization is necessary to achieve performance gain. In parallel programming the platforms of the memory hierarchies and the processors in the system are different. This makes the program optimization more complicated. Unlike sequential programming, the program optimization in parallel programming depends on understanding the hardware of the system and estimating errors at every step of execution. The optimized program must be tested at every step by experimenting with all possible error prone situations. [13]

Testing and debugging techniques in parallel programming are much different compared to sequential programming. As parallel programs are very complex, clear knowledge on program execution is needed while designing the testing cases. Detailed testing and debugging techniques are developed, as the processors and memory hierarchy platforms are different. [13]

Adding new features to parallel programming applications is difficult and involves more risk. As there are chances of affecting the total performance of the system, more care should be taken while adding new features to the program. [13]

5. SCHEDULING IN MULTIPROCESSOR SYSTEM

Scheduling is the technique of allocating processor time to processes in the tasks for execution. Scheduling helps for executing the list of given tasks efficiently and smoothly. In order to determine the sequence of execution of the processes in the tasks, a well-defined scheduling algorithm is followed. [1]

Initially, all the input processes in the tasks are fed to a ready queue. A scheduler selects the processes from the ready queue to allocate processor time for execution. A dispatcher dispatches the process to the processor. This is explained in Figure 5.1. [1]

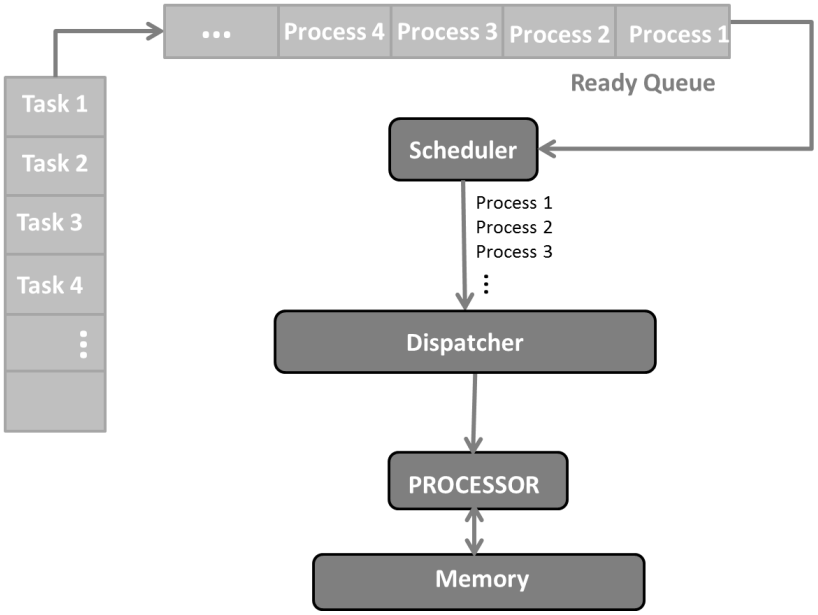

Figure 5.1 Process allocation

5.1 Scheduling Algorithms

The scheduling is implemented using scheduling algorithms. Based on the kind of task, the kind of processes in the task and type of system, appropriate scheduling algorithm is chosen. The following are few well know scheduling algorithms. [1]

5.1.1 First In First Out Scheduling Algorithm

First in first out scheduling algorithm schedules the processes to be executed in the same order in which they arrive. Figure 5.2 describes the First in First out scheduling algorithm [1]

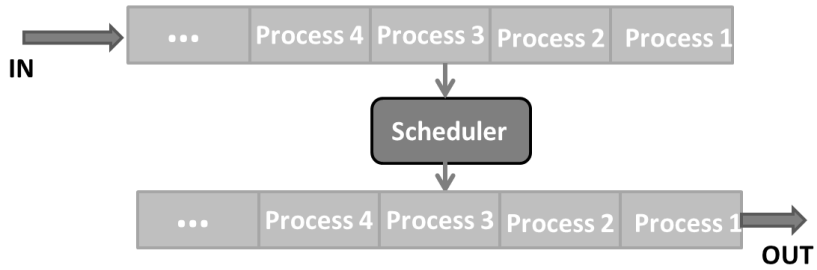

Figure 5.2 First in First out scheduling

5.1.2 Shortest Job First Scheduling Algorithm

Shortest job first scheduling algorithm schedules the processes based on the estimated time taken to execute them. The shortest time taking process will be executed first. Figure 5.3 explains the shortest job first scheduling.

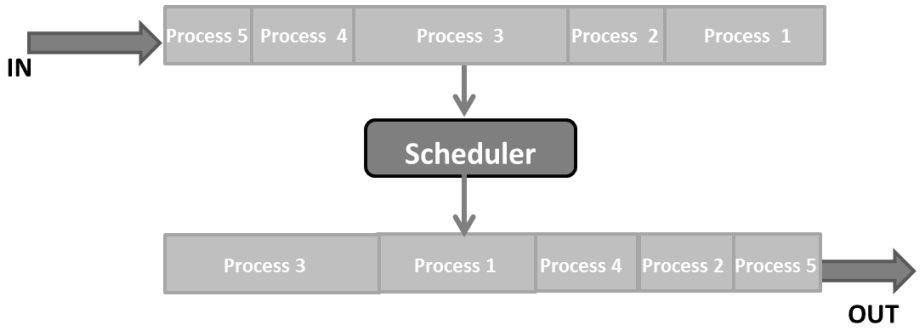

Figure 5.3 Shortest job first scheduling

5.1.3 Priority Based Scheduling Algorithm

In priority based scheduling algorithm, each process is assigned a priority externally. Based on these priorities, the processes are scheduled. The highest priority process is scheduled to be carried out first and so on for the other processes. Figure 5.4 explains the Priority based scheduling. [1]

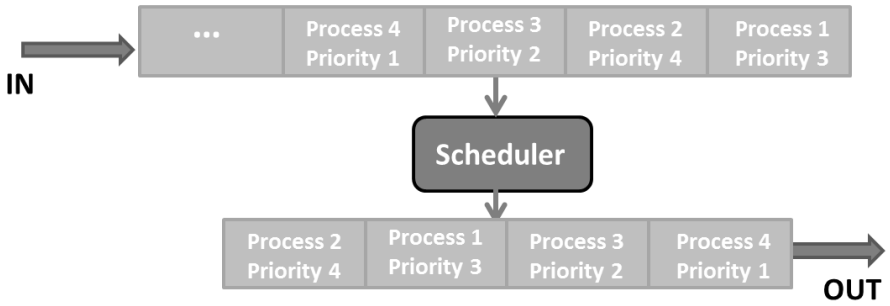

Figure 5.4 Priority based scheduling

5.1.4 Round Robin Scheduling Algorithm

In round robin scheduling algorithm, all the processes in the queue are executed for a fixed unit of processing time in a cycle. The CPU time is fairly distributed between all the processes. Each process gets equal share of CPU time. This allocation of CPU process time is done in a cyclic way. Once the process is executed, it gets eliminated from the queue. Figure 5.5 explains Round Robin scheduling. [1]

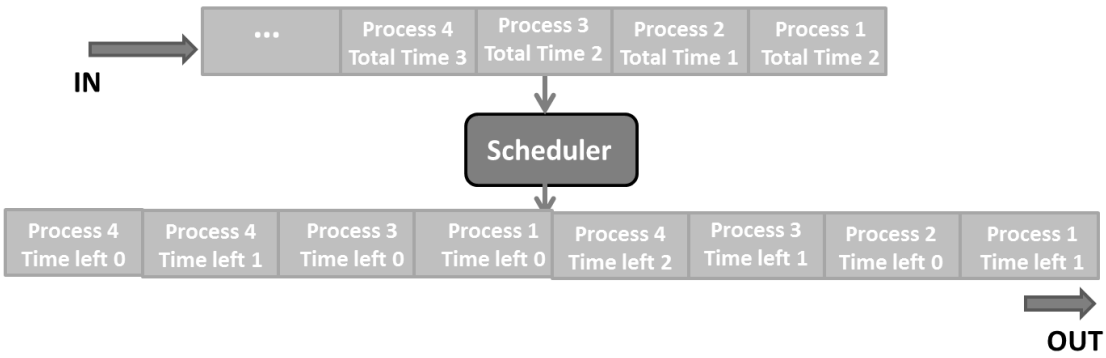

Figure 5.5 Round Robin scheduling

5.1.5 Multilevel Queue Scheduling Algorithm

In multilevel queue scheduling algorithm, all the processes are classified into different groups according to the type of task performed. Each group of processes is fed to a different queue. Each queue is scheduled differently with an appropriate scheduling technique. All the queues are executed in parallel. [1]

5.2 Multiprocessor System Scheduling

The scheduling in multiprocessor systems is the execution of all the processes in the given tasks on set of existing processors under optimizing criteria. The goal of scheduling is to minimize the run time of a task set. [1] [18]

5.2.1 Scheduling Criteria

Scheduling criteria is decided based on the type of the system, efficiency and number of processors, kind of tasks executed, interaction and dependency between the processes in the task and sharing of resources among the processes. The general optimizing scheduling criteria includes:

- Processor utilization: Processor should be as busy as possible, that means, processor utilization should be high.
- Throughput: Throughput is the number of processes completed in unit time. Throughput should be high.
- Turnaround time: Turnaround time is the time taken to execute a process. Turnaround time should always be low.
- Waiting time: Waiting time is the amount of time that a process spends waiting in the Ready queue. Waiting time should be low.
- Response time: In an interactive system, the time from the submission of the request to the first response produced is called the response time. The response time should be as low as possible.
- Deadline handling: Deadline handling is the ability to execute a process in a given time. Deadline handling should be performed.
- Starvation free: Starvation is the condition of a waiting process which is never executed. The scheduling algorithm should be starvation free. [1] [18]

Other scheduling criteria include minimizing the cost, minimizing communication delay, and giving priority to certain processes. The scheduling policy for a multiprocessor system usually involves these criteria. [3]

Pre-emptive and non-pre-emptive scheduling schemes must be specified in the scheduling algorithm. In pre-emptive scheduling, based on predefined pre-emptive conditions, the scheduler can pre-empt or kill a process which is taking too long time to execute. In non-pre-emptive scheduling the scheduler allocates the processor to the process till it is completely executed. [1]

5.2.2 Implementation of Scheduling in Multiprocessor Systems

Scheduling in multiprocessing systems is implemented based on the architecture of the system. Hardware constraints such as sharing resources, speed of bus, communication, coupling of the processors and memory access will influence the scheduling. Also, software constraints such as data and instruction processing also affect the implementation of scheduling in multiprocessor systems. The two major approaches to multiprocessor scheduling are Asymmetric multiprocessing and Symmetric multiprocessing. [1]

In asymmetric multiprocessing, the processors follow Master-Slave mechanism. Master processor decides the scheduling and other processors follow instructions from the master processor and execute the processes. Master processor distributes the processes from the ready queue among the processors based on the scheduling algorithm. [1]

In symmetric multiprocessing, execution at each processor is independent of one another. Here scheduling is carried out using 2 levels of queues − a global ready queue and a local ready queue at each processor. [3] [1]

All the processes in the given tasks are loaded into the global ready queue initially. Based on the predefined scheduling criteria of the system, all the processes are rearranged. From the global ready queue, rearranged processes are then moved to the local private queues at each processor in the system. This way, the load is shared between the processors in the system. At each processor private ready queue, the processes are executed based on the scheduling algorithm. Figure 5.6 explains scheduling in symmetric multiprocessor system. [3] [1]

To speed up the total execution time, processes at busy processors can be transferred to idle processors. [3]

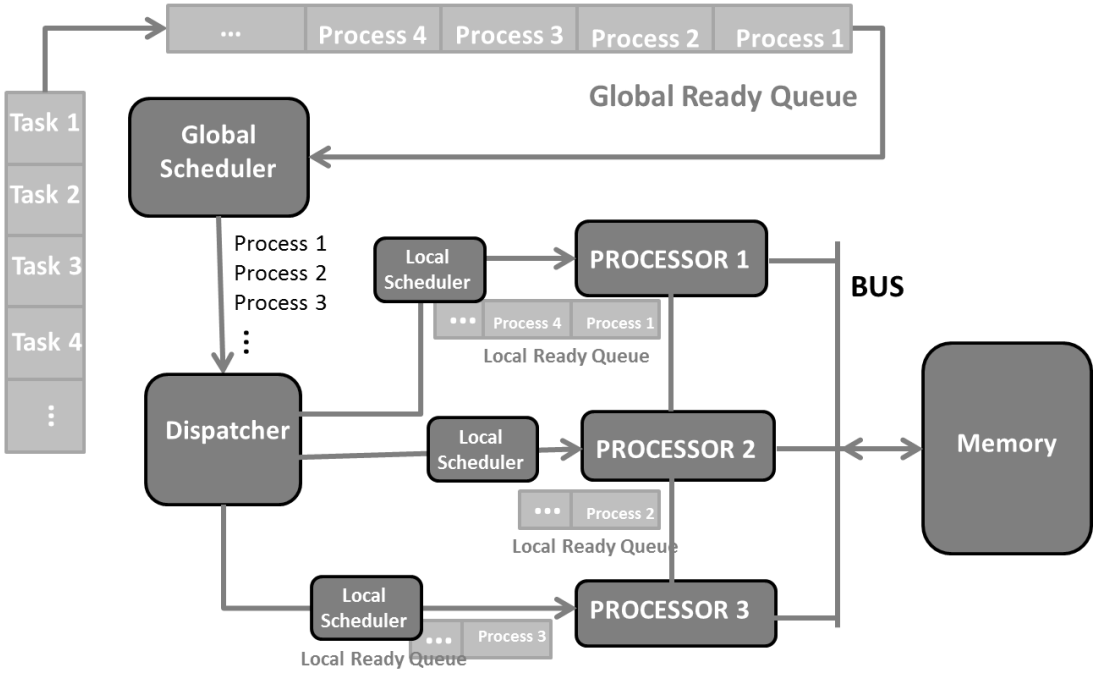

Figure 5.6 Scheduling in multiprocessor systems

Communication between the processors can be carried out by message passing or by using shared memory. Resource sharing is carried by maintaining critical sections and locks. Whenever a process needs to use a shared resource, then to access the resource the process enters the critical section and locks the resource. After finishing the execution the process comes out of critical section and releases the lock. During this time, no other process can access that resource. If a process takes too long time to release a resource, then based on pre-emption conditions the process can be pre-empted.

The stability of a system is determined by the speed at which tasks arrive and the speed at which the task complete. In general, as the system runs, new tasks arrive while old tasks complete execution. A system is said to be unstable if the speed at which the tasks arrive is greater than the speed at which they are executed. A system is said to be stable if the speed at which the tasks arrive is lower that the speed at which they are executed. A stable scheduling policy will never make a stable system unstable. Unstable scheduling policy can push the system into instability if the arrival rate is higher than the service rate for a system. [3]

5.2.3 Choosing a Scheduling Algorithm

Choosing an appropriate scheduling algorithm is necessary in order to complete a given task efficiently. There is no universal best scheduling algorithm. In practical, an appropriate scheduling algorithm for the system is chosen based on the

kind of job, dependencies among the processes and the resources needed by the processes to execute. [1]

In many cases, two or more scheduling algorithms can also be combined to get the desired algorithm. If the processes in a task are not scheduled properly then the task is less likely to get executed. An appropriate process scheduling algorithm for a given job is the one which fulfils the scheduling criteria of the system. [1]

5.2.4 Problems in Multiprocessor Scheduling

The following are the major issues in multiprocessor scheduling:

- Allocation and sequencing: Analysing the sequence of operations of the processes at processors is difficult when an input for execution of one process is dependent on the output of other process which is executed at a different processor. In such cases the final result depends on how well the processes are synchronised.

- Load balancing: All the processes which are fed to processors for execution are not of same size. Predicting the size of continuous input load in the system is not possible. Thus, load balancing among all the processors in the system is not 100% possible.

- Communication delays and network delays: These delays may lead to miscommunication between the processes and may result in errors in solution. Also these delays give wrong estimation of execution times.

- Loop scheduling: Predicting the actual loop execution time and related work overhead while distributing and executing loops among different processors in parallel is difficult. [19]

6. SIMULATION IN MULTIPROCESSOR SYSTEMS

Simulation of multiprocessor system is an abstract representation of the proposed or existing system as a software program. Simulation helps to understand the behaviour of the multiprocessor system. [37]

The simulation model is built by replicating the functionality of the multiprocessor system design, hardware functionality and the software model. The inputs and the data such as clock cycles should be as accurate as possible to drive the simulation and to calculate the outputs accurately. The results of the simulation model are represented using graphics, charts or data. [1][37][38]

Multiprocessor systems are very complex. To develop simulation models for such systems accurately, all the factors influencing and affecting the system should be carefully studied and considered. Hence, accuracy and validity of the simulation model compared to the actual system depend on factors influencing system considered while modelling. The drawbacks or functional bugs of the actual system can also be identified by experimenting on these simulation models. [37][38]

Today, the size of simulation of multiprocessor systems vary from small to huge which run from minutes to hours. These simulations find extreme need in various fields such as computers, networking, atmospheric dispersion modelling, logistics, noise mitigation, flight simulators, weather forecasting, reservoir simulation, robot simulators, traffic engineering and more. [33]

The major issues associated with simulating these complex multiprocessor systems are accuracy and reproducing results. As there will be many real world factors influencing the actual system, it is very difficult to predict and consider all such factors while simulating the system. Hence simulations are not 100% accurate to the actual systems. Reproducing the same results is also not possible in the cases where random numbers are used in inputs or in calculation of intermediate values in the simulation. Thus verification and validation of the simulation results is very crucial. [33]

As the clock cycles, network delays and real world problems in the actual system are very difficult to predict and calculate, the multiprocessor systems simu-

lation models accuracy is always questionable. In spite of such accuracy issues, simulation of multiprocessor system is always helpful to study, understand and analyse the actual system. [37][38]

7. DESIGN AND IMPLEMENTATION OF SIMULATION MODEL FOR MULTIPROCESSOR SYSTEM SCHEDULING

A simulation model is designed and implemented for understanding the working of multiprocessor system scheduling. This simulation model consists of symmetric processors, which means that all the processors are treated equally. Depending on the input efficiency of the processors and the process priority, processes are distributed among the processors equally and they are allocated processor time for execution. The total execution time of the input job is calculated. Pre-emption of processes is also handled based on predefined conditions.

An XML parser is integrated with this system to extract the inputs from XML files to text files.

Message passing among the processes, resource allocation and sharing, networks delays in the system, task preferences in the input job and load balancing among the processors are not considered in the design of this simulation model for multiprocessor system scheduling.

7.1 Scheduling Criteria in the Simulation Model

The scheduling criteria of the system includes

1. Load sharing: Depending on number and efficiency of the processors, all the processes of the input tasks are distributed among all the processors. Hence, the load is shared among all the processors in the system.

2. Processor utilization: Processors are put as busy as possible by assigning the processes to all the processors for execution (This factor depends on amount of load on the system).

3. Priority considerations: Priority of the processes is considered while execution. All the processes are scheduled for execution according to their priority.

4. Process pre-emption: The processes which take lot more than expected time to execute are pre-empted following pre-define conditions. Pre-emption conditions are checked for each process at every clock cycle.

5. Starvation free: At every clock cycle the processes are checked to know if any process is starving for the execution. When found, it is put to processor ready queue and allocated processor time for execution immediately.

6. Waiting time: The waiting time of the processes is controlled by raising the priority of processes that have been waiting a long time to be executed.

7. Response time: As the load is shared among the processors in the system, the execution is faster and response time is reduced.

8. Turnaround time: As the processes are loaded in to the processor queue according to processor efficiency, the time taken to execute a process will be lower. Hence, the turnaround time is put to as low as possible.

7.2 Concept Model

The concept model gives the overview of the system. Figure 7.1 illustrates the concept diagram of simulation of multiprocessor system scheduling design.

From the concept diagram, it can be seen that the tasks are fed to the system through an XML file. An XML parser is used to parse the processes information of the tasks from the XML file. Then the processes' information such as time needed to finish the process, process priority and process number are given as the input to the system in a text file. Also the processor information is fed to the system. This processor information can also be modified through the GUI of the system. The processor information contains the processor number and its efficiency. According to the processor efficiency, each processor capacity is calculated. The scheduler schedules the processes by distributing the processes among the processors where the processes are executed.

During the execution, the system displays errors messages (such as file read write errors, file missing errors), process pre-emption messages and finally the total execution time taken to finish the job.

Apart from execution, system also provides operations such as estimate, reset and exit. Estimate operation, estimates the total time that would take for the system to finish the job and gives total estimated execution time as result. Reset helps to reset the system by erasing all the existing inputs, outputs and intermediate data. Exit operation is used to exit from the system at any time.

The display of errors messages (such as file read/write errors, file missing errors) and intermediate process pre-emption messages helps the user to understand the execution in the system.

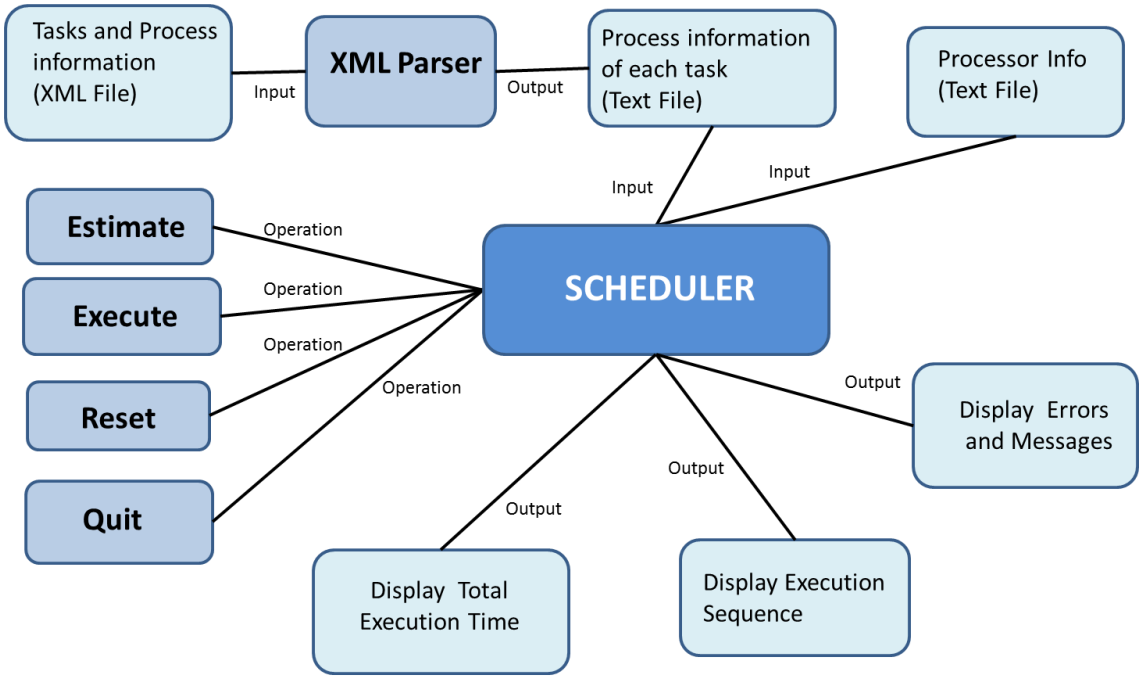

Figure 7.1 Concept diagram

7.3 Design of Scheduler

The scheduling and internal working of the scheduler of the system is described in detail here. The Flow diagram in Figure 7.2 shows the follow of execution in the scheduler of the simulation.

After filtering the processes' information from the XML file, at global level the processes are fed to a waiting queue. According to the process priority, the processes are rearranged in the waiting queue. Also, based on the input processor information, the capacity of the processors is calculated. At each processor a local ready queue is maintained. These local ready queues at each processor are loaded with the processes in the global waiting queue. At each processor a scheduling algorithm is followed to execute the processes in its ready queue.

Priority based Round Robin scheduling is used in this simulation model to execute the processes in the ready queue at each processor.

During execution, at each clock cycle, the time required to execute the process (time stamp of the process) is reduced by a predefined value. When the time stamp of the process becomes zero, the process execution is completed and is removed from the respective ready queue. After removal of the process, the next priority process from the waiting queue is loaded. If waiting queue is empty, then no process will be loaded. Also, at every clock cycle the pre-emption conditions are checked for all the processes in the waiting queue and in all ready queues at each processor.

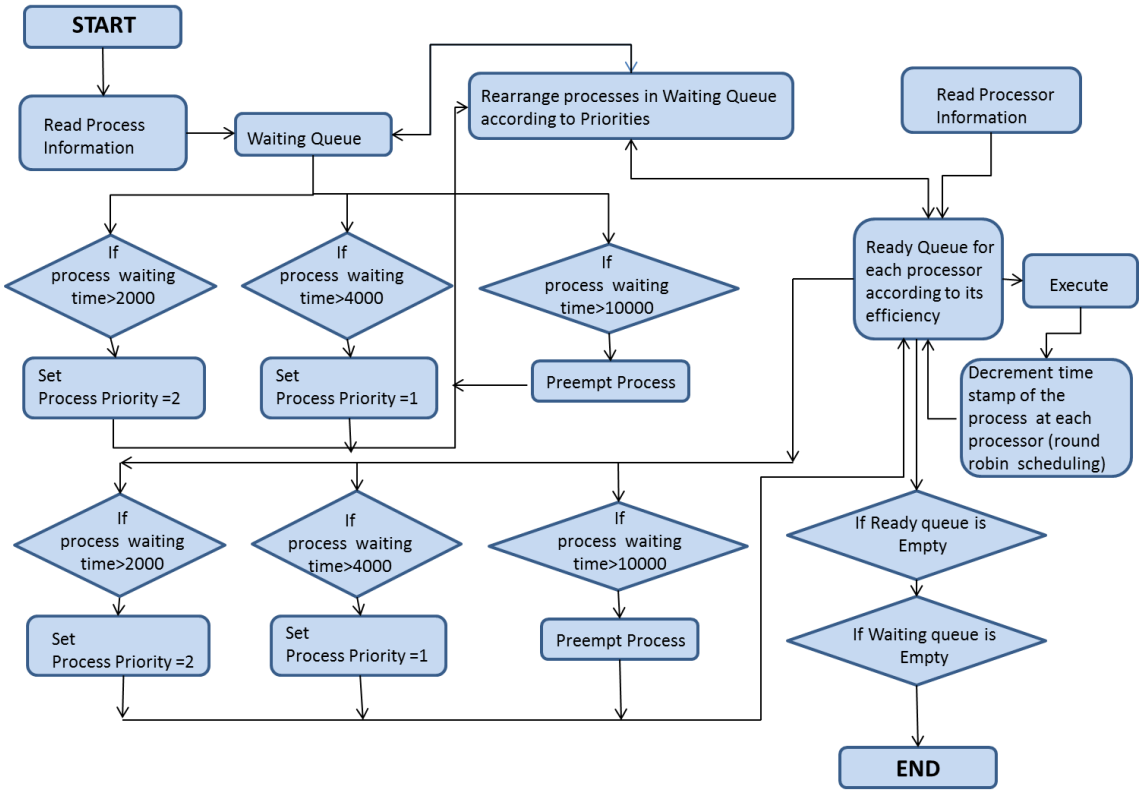

Figure 7.2 Flow diagram of scheduler

If a process is found waiting for a long time, then the process is given higher priority according to predefined pre-emption conditions. If the process execution time or waiting time takes even longer (longer than the predefined limits), then the process is pre-empted or killed. The given job completes only after all the processes in the ready queues and waiting queue are executed.

7.4 Example

The step by step execution inside the simulation model is explained using an example in this section. Figure 7.3 shows the processes and processor information, which are the inputs to the scheduler in this example. This is the processes information after extracting from the XML file using XML parser. The processor efficiency values in this example are the calculated values of the actual input.

Processors Information

Processor Number	Efficiency
1	3
2	2
3	1
4	2

Processes information in a task

Process Number	Time required for executing each process	Priority of each process
1	4	3
2	3	2
3	5	1
4	2	2
5	3	1
6	1	3
7	2	1

Figure 7.3 Inputs

From the processor's information, each processors capacity to execute number of processes is determined from its efficiency. As can be seen from Figure 7.3, processor 1 can accommodate and execute 3 processes at a time, processor 2 can accommodate and execute 2 processes at a time, processor 3 can accommodate and execute 1 process at a time and processor 4 can accommodate and execute 2 processes at a time.

From processes' information, the priority and required execution time of each process are extracted. Based on processor capacity, processes are distributed

among the processors to execute in the order of their priorities. This distribution of processes among processors is shown in Figure 7.4.

Based on processor and process information in Figure 7.3, as the processes 3, 5, 7 have priority 1 and they are allocated processor time first. Hence, process 3 with time required for execution - 5 time units is fed to processor 1. Process 5 with time required for execution - 3 time units is fed to processor 2. Process 7 with time required for execution - 2 time units is fed to processor 3. After distributing processes with priority 1 to the processors, processes with priority 2 are fed to processors. This distribution of processes' among processors is done till all ready queues are full or there are no more processes to execute.

	TimeUnit-0		TimeUnit-1		TimeUnit-2		TimeUnit-3		TimeUnit-4		TimeUnit-5		TimeUnit -6		Time Unit-7	
	Process/ Time required		Process/ Time required		Process/ Time required		Process/ Time required		Process/ Time required		Process/ Time required		Process/ Time required		Process/ Time required	
Processor1	3/5	4/2	3/4	4/2	3/4	4/1	3/3	4/1	3/3	4/0	3/2		3/1		3/0	
Processor2	5/3	1/4	5/2	1/4	5/2	1/3	5/1	1/3	5/1	1/2	5/0	1/2	1/1		1/0	
Processor3	7/2		7/1		7/0											
Processor4	2/3	6/1	2/2	6/1	2/2	6/0	2/1		2/0							

Figure 7.4 Process execution sequence

The Figure 7.4 shows the execution sequence of the processes. Each processor follows Round Robin scheduling to execute the processes in its ready queue. As shown in the Figure 7.4, the processes 3, 5, 7 and 2 are executed in the first clock cycle at time unit 0. As they are executed, the time required to execute the respective process is reduced by 1. At time unit 1, the processes 4, 1, 7 and 6 are executed. At time unit 2, process 3, 5 and 2 are executed. As the process 7 has finished its execution, it is removed from the ready queue. As no other process is waiting for execution, no process is loaded in place of process 7. This way the execution continues and by time unit 7, all the processes are completed and the job is finished.

7.5 Technologies Used

The Table 7.5 illustrates the technologies used in developing the simulation model.

Languages	**Implementation**	SystemC – Version 2.2
	Graphical user interface	QT- Version 4.5
Input data	XML files	
Framework	Microsoft Visual Studio 2008	
Platform	Windows XP	

Table 7.5 Technologies used

7.6 Graphical User Interface, Inputs and Outputs

7.6.1 Graphical User Interface

The graphical user interface used in this simulation model is shown in Figure 7.6

The input process data XML file name and its path are entered in the text box opposite to label “Enter XML file name”. The processor information in text file name and its path are entered in the text box opposite to label “Enter processor info file name”. The processor information is displayed using “Display processor file contents” button in the respective text area.

The processor information can be modified in the text area and updated in the file using “update processor info file contents”. The errors and pre-emption messages during execution are displayed in “Errors or messages” text area. The total execution time taken to complete the job is displayed in the text box “Total execution time”.

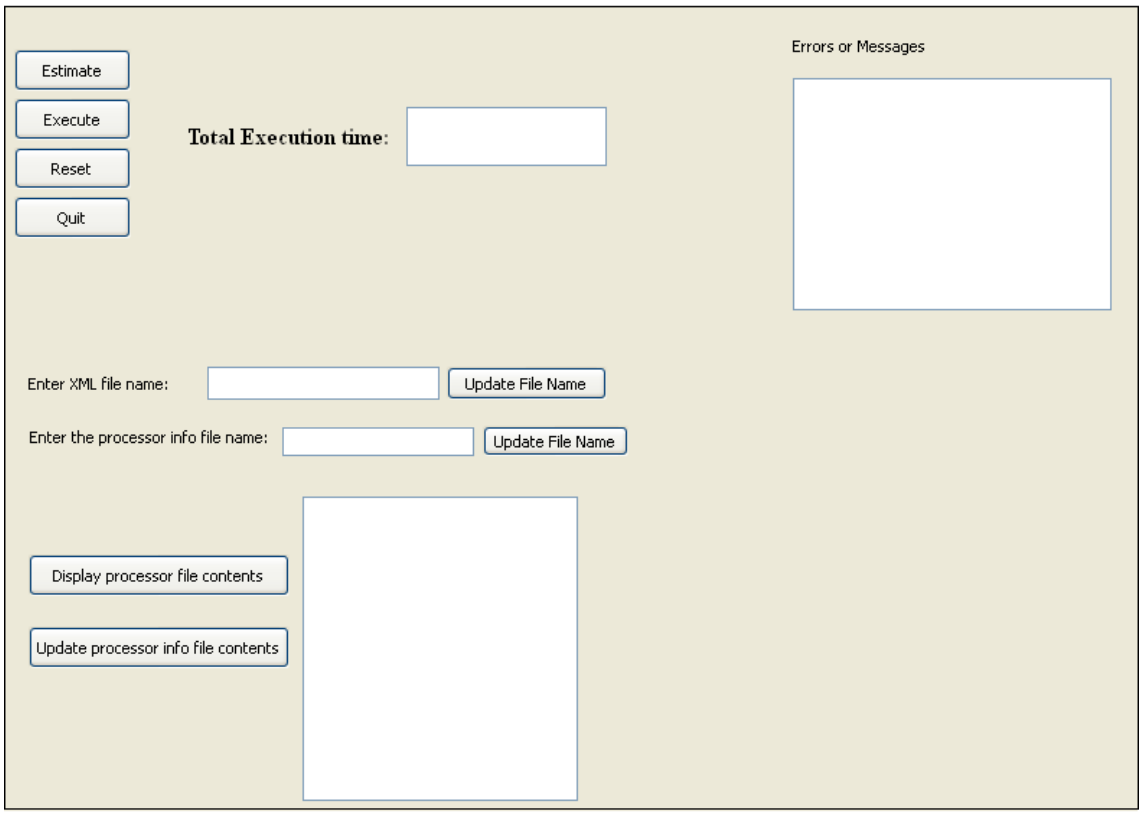

Figure 7.6 Graphical user interface

"Estimate", "Execute", "Reset" and "Exit" are the operations. "Estimate" operation is used to estimate the total time to complete the job. "Execute" operation is used to execute the job. "Reset" operation clears all the text boxes, text areas in GUI and input, output files. "Quit" operation exits the application any time.

7.6.2 Inputs

The following are the inputs to the simulation model:

- An XML file containing information of the tasks and their respective processes such as name of the tasks, process ID, process preference, required process execution time and process priority.
- A Text file containing processors information such as processor ID and processor capacity.

7.6.3 Outputs

The following are the outputs of the simulation model:

- Estimation of total execution time
- Total execution time taken to finish the job

- The errors and pre-emption messages during execution. Figure 7.7 shows an example for display of Pre-emption Messages in GUI.

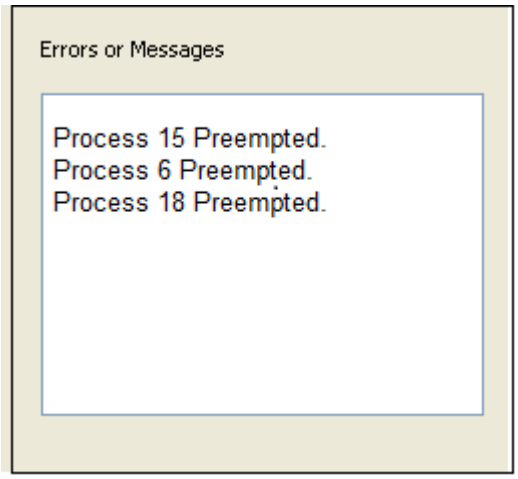

Figure 7.7 Pre-emption Messages

All the outputs are displayed in the GUI and also stored in output files.

Most important modules of the implementation of the simulation model of multi-processor scheduling system is in appendixes.

7.7 Result Analysis

The results of the simulation of multiprocessor system scheduling are analysed in this section.

7.7.1 Analysis 1

Input Conditions
Job information: Same job is executed in all the experiments

Processor information: Capacity of all the processors is constant, which is 1. Number of processors is increased by 1 every time, starting from 1 to 8.

Experiment Result
With the increase in number of processors in the system, the total execution time decreases. But after a point the system achieves saturation limit where the increase in the number of processors in the system does not affect the total execution time. This is shown in the Figure 7.8 Total execution time versus number of processors.

In the Figure 7.8, at the beginning, by increasing the number of processors from 1 to 5, very slight decrease in execution time is noticed, this is due to heavy load on the processors. By increasing the number of processors to 6, a drastic decrease in execution time can be seen. This drastic decrease in execution time is because of the reduced load on all the processors. But after increasing the number of processors from 6 to 8, there is no change in total execution time. At this point system achieves a saturation limit.

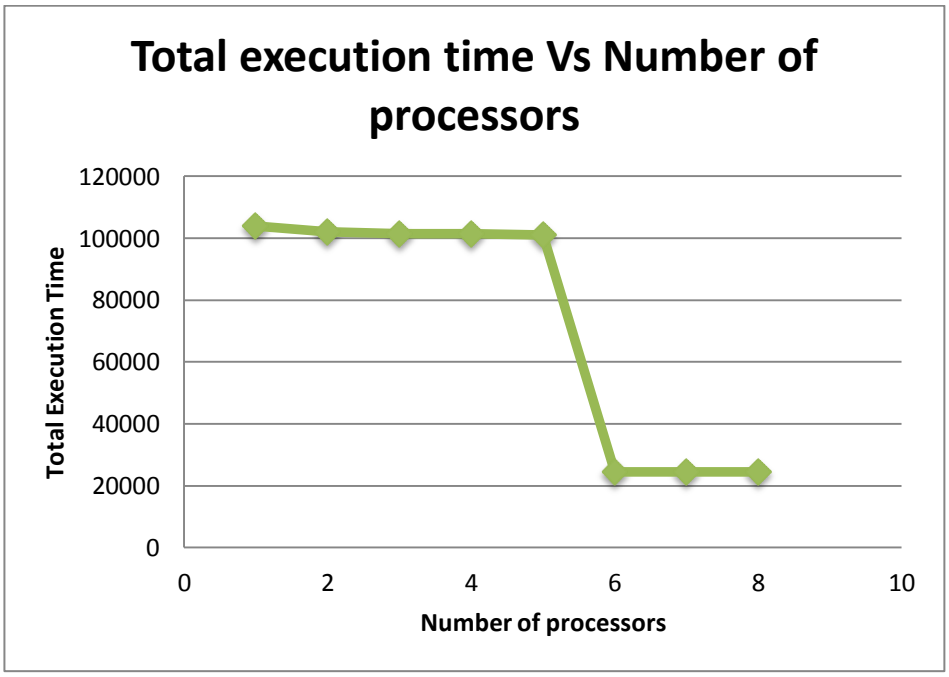

Figure 7.8 Total execution time versus number of processors

The saturation limit varies with the size of the job being executed. The load is distributed among all the processors in the system. The maximum number of processors required to execute the job is limited to the size of the job. When the number of processors exceed beyond the maximum limit, the extra processors stay idle. Hence after this saturation limit, the increase in number of processors in the system does not affect the total execution time of the job.

7.7.2 Analysis 2

Input Conditions

Job information: Same job is executed in all the experiments

Processor information: capacity of all the processors is increased by 1 every time, starting from 3 to 10. Number of processors in the system is constant (3 processors) for all the experiments

Experiment Result

The increase in the capacity of the processors in the system will decrease the total execution time to an extent, when the number of processors in the system is constant while executing same job every time. This is shown in Figure 7.9 Total execution time versus capacity of processors.

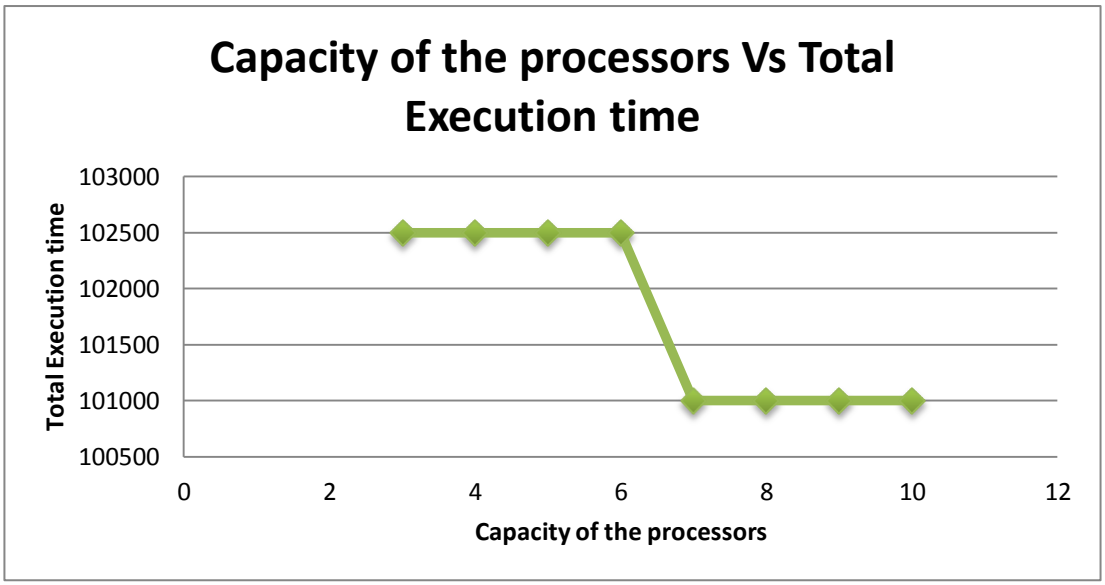

Figure 7.9 Total execution time versus capacity of processors

With the increase in the capacity of the processors each time, the processors will be overloaded and will reach a saturation limit. After that limit, the increase in capacity of the processors does not affect the speed and efficiency of the system. At that point, the increase in number of processors can decrease the total execution time by sharing the load.

8. CONCLUSION AND FUTURE WORK

This chapter presents the main conclusions of the thesis and the future work. In this thesis, the major focus is to understand how the overall performance of multiprocessor systems is improved by scheduling and sharing the workload among all the processors in the system. Also, the change in speed of the multiprocessor system with the change in capacity of the processors and the number of processors is analysed.

Compared to uniprocessor systems, the throughput and speed of the multiprocessor systems is high when an appropriate scheduling algorithm is used for executing the input tasks. The appropriate scheduling algorithm is chosen based on the tasks executed and the type of the system. An ideal scheduling algorithm reduces the total execution time, increases the throughput, reduces the waiting time of subsequent tasks in the queue, maximizes the processes resource utilization and allocates processor time for each process equally.

In cases, when the input is very low, then the uniprocessor systems are faster than the multiprocessor systems. This speed difference is because of the extra network transfer time which is added to the total execution time in the multiprocessor systems.

In spite of accuracy issues, the experiments done within this thesis demonstrated the importance of process scheduling in increasing the execution speed and efficiency in the multiprocessor systems.

In the future, to increase the accuracy of the simulation model of multiprocessor system scheduling and to calculate the total execution time more accurately few more factors must be considered. Those factors include

- Networks delays in the system
- Task preferences in the input job
- Message passing among the processes
- Resource allocations to the processes
- Load balancing among the processors by transferring processes from one processor to other during the execution.

REFERENCES

1. Abraham Silberschatz, Peter Galvin, Greg Gagne. Ninth Edition. *Operating System Concepts*. John Wiley & Sons, Inc, Hoboken, NJ. Pages 14-16, 105-147, 203-258, 261-304, 315-337.
2. Blaise Barney, Lawrence Livermore National Laboratory. *Introduction to Parallel Computing.* https://computing.llnl.gov/tutorials/parallel_comp/#Whatis
3. Steve J. Chapin, Jon B. Weissman. (January 2002). *Distributed and multiprocessor scheduling*. Syracuse University, University of Minnesota. http://www-users.cs.umn.edu/~jon/papers/handbook.pdf
4. Tim M. Jones. (March 2007). *History of multiprocessing*. IBM. http://www.ibm.com/developerworks/library/l-linux-smp/
5. Noora sadon, Rand Nawfal, Karar Shakir, Muhanned Raad. Retrieved on January 5, 2011. *Multiprocessing System*. Itswitech University. http://www.itswtech.org/Lec/Manal(system%20programming)/simeners_A/Multiprocessing_System_Cimenar.pdf
6. Ben-Ari, Mordechai (2006). *Principles of Concurrent and Distributed Programming (2nd ed.)*. Addison-Wesley.
7. Gregory V. Wilson. (October 1994). *The History of the Development of Parallel Computing.* Toronto University. http://ei.cs.vt.edu/~history/Parallel.html
8. Mike Pacifico, Mike Merril. (1998). *A General Overview of Parallel Processing.* CMSC 411 Project. http://www.cs.umd.edu/class/fall2001/cmsc411/projects/parallel2/history.html
9. Herb Sutter and James Larus. (September 2005). *Software and the concurrency revolution.* Microsoft.
10. SAS group. Retrieved on January 20, 2011. *Independent Parallelism*. SAS Institute. http://support.sas.com/documentation/cdl/en/connref/61908/HTML/default/a002626620.htm

11. IBM Corporation. (March 2006). *Locks and concurrency control.* http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/c0005266.htm
12. Dave Thomas, Chad Flower, Andy hunt. 2009. *Programming Ruby - The Pragmatic Programmer's Guide.*
13. Wen-mei Hwu, Timothy G.Mattson and Kurt Keutzer. (August 2008). *The Concurrency Challenge*. Urbana-Champaign, Intel, Berkeley.
14. David Geer. September 2007. *For programmers, Multicore chips mean multiple challenges*. Freelance technology, Ashtabula, Ohio.
15. Arvind, Robert A lannucci. Laboratory for Computer Science. Massachusetts Institute of Technology. *Two Fundamental Issues in Multiprocessing.* http://webcourse.cs.technion.ac.il/236604/Spring2012/ho/WCFiles/limits-MP.pdf
16. Saleh Elmohamed. 2002. *Parallel processing concepts.* CTC. http://mpc.uci.edu/wget/www.tc.cornell.edu/Services/Edu/Topics/ParProgCons/
17. Alan Joch. (November 2000). *Chip Multiprocessing*. http://www.computerworld.com/s/article/54343/Chip_Multiprocessing
18. Siber cankaya. *CPU scheduling*. http://siber.cankaya.edu.tr/OperatingSystems/ceng328/node1.html
19. Selected Problems of Scheduling Tasks in Multipro cessor Computer Systems. By Maciej Drozdowski
20. Toby Foster, Electronic Design. *Symmetric Multiprocessing Vs. Asymmetric Processing.* http://electronicdesign.com/digital-ics/symmetric-multiprocessing-vs-asymmetric-processing
21. Siber cankaya. *Flynn's Taxonomy of Computer Architecture.* http://siber.cankaya.edu.tr/ParallelComputing/ceng471/node14.html
22. Jonathan Appavoo. *Optimizing Multi-Processor Operating Systems Software Research Review.* http://www.cs.bu.edu/~jappavoo/Resources/Papers/depth.pdf
23. David Kanter. *An Introduction to multiprocessoe systems*. December 11, 2006. http://www.realworldtech.com/coherency/

24. Gordon Bell. *Three Decades of Multiprocessors*. http://research.microsoft.com/en-us/um/people/gbell/cgb%20files/three%20decades%20of%20multiprocessors%20acm%201991%20c.pdf
25. Gregory V. Wilson. (October 1994). *The History of the Development of Parallel Computing.* Toronto University. http://ei.cs.vt.edu/~history/Parallel.html
26. Mike Pacifico, Mike Merril. (1998). *A General Overview of Parallel Processing.* CMSC 411 Project. http://www.cs.umd.edu/class/fall2001/cmsc411/projects/parallel2/history.html
27. Leading Electronics INC. *Moore's Law or how over all processing power of computers will double every two years*. http://www.mooreslaw.org/
28. Intel official web page. *Moore's Law and Intel Innovation.* Extracted in February 2014. http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html
29. Princeton. *Amdahl's law.* https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Amdahl_s_law.html
30. Oderog. *Super Computers, Micro computer , Mini computers, and Mainframe Computers.* http://oderog.hubpages.com/hub/supercomputers--micro-computer---mini-computers--and-main-frame-computers
31. Kunle Olukotun and Lance Hammond. (September 2005). *The future of microprocessors*. Stanford University.
32. Barrett-group. Mcgill. *Nanotechnology: A Brief Overview.* http://barrett-group.mcgill.ca/tutorials/nanotechnology/nano03.htm
33. Wikipedea. *Computer simulation.* Retrieved on 25.01.2014
34. Nguyen Thi Hoang Lan. *Advanced Architectures.* http://cnx.org/content/m29689/latest/
35. Oracle coorporation. *Parallel Hardware Architecture.* http://docs.oracle.com/cd/A57673_01/DOC/server/doc/SPS73/chap3.htm

36. Eurípides Montagne. University of Central Florida. *Lecture: Flynn's Taxmony.* Retrieved on 1st February, 2014.
37. Roger D. Smith. 1998. *Simulation Article.* http://www.modelbenders.com/encyclopedia/encyclopedia.html
38. Sebastien Nussbaum, James E. Smith. *Statistical Simulation of Symmetric Multiprocessor Systems.* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.1101&rep=rep1&type=pdf
39. DocStoc. *Introduction to computer simulation.* http://www.docstoc.com/docs/2690471/Introduction-to-Computer-Simulation-Uses-of-Simulation
40. Charles J.Marcraft. Pearson publication. *Server+* . http://my.safaribooksonline.com/book/certification/serverplus/9780768690200/multiprocessing-systems/ch03lev2sec3

APPENDIXES

This section contains the most important modules of the implementation of simulation model for multiprocessor system scheduling.

Simuation_scheduling.h

```
///*************************************************************************************************
//Header file for structure of the Simulation model for multiprocessor //scheduling
///*************************************************************************************************

#include "systemc.h"
#include <conio.h>
#include <string>


SC_MODULE (scheduling)
{
public:
SC_CTOR (scheduling)
{
}
void initialise();
void read_pr_info();
void read_process_info();
void wq_to_rq();
void swap_acc_to_priority();
int pr_execution(char status);
void print();
private:
// Process Data Structure
struct Processdata
{
int processnumber;      // Process Number
std::string sequenceName; // Sequence Name
int preference;             // Preference
unsigned int processtimestamp; // Timestamp
int processpriority;        // Priority
unsigned int processwaitingtime;  // Waiting Time
int packetcnt;              // Packet count
char processprempted;  // Preemption variable
}process;

// Processor Data Structure
struct Processordata
{
int processor_num;        // Processor Number
int processor_efficiency; // Processor efficiency
int processorwaitingtime; //waiting time of processes
}processor;

Processdata readyqueue[30][30]; // Ready queue
Processdata waitingqueue[50]; // waiting queue
Processdata disp[50];    // completed processes values
Processdata dtemp[30]; // Temprory values
```

```
Processdata tmpar; // temprory swaping variable
Processdata modifiedprocessqueue[50];

Processordata dpr[30];    // processor data

FILE *pptr;          // file pointer for input processes values
FILE *prptr;  // file pointer for input processores values

// Count Variables
int npwq;          // count total number of process in the waiting Q
int np_tot;        // Temp total number of process in waiting Queue
int nrq1[20]; // count total number of process in the ready Q

int npr;  //  number of processors
int nprct[50];      // number of processors in each processor
int mod_pr_num;
char status;       //Variable to hold the command
std::string fname;  //variable to hold filename
std::string pr_fname;  //variable to hold processor filename
};
```

Simuation_scheduling.cpp

```
///*****************************************************************************
//Functions of scheduling in multiprocessor system of simulation model
///*****************************************************************************

#include "simuation_scheduling.h"

// function to initialise all the values
void scheduling::initialise()
{
// constructor
npwq=0;
np_tot=0;
npr=0;
mod_pr_num=0;

for(int i=0;i<=30;i++)
{
for(int k=0;k<=30;k++)
{
readyqueue[i][k].processnumber=0;
readyqueue[i][k].processtimestamp=0;
readyqueue[i][k].processpriority=0;
readyqueue[i][k].processwaitingtime=0;
readyqueue[i][k].processprempted='N';
}
}
for(int k=0;k<=50;k++)
{
waitingqueue[k].processnumber=0;
waitingqueue[k].processtimestamp=0;
waitingqueue[k].processpriority=0;
waitingqueue[k].processwaitingtime=0;
waitingqueue[k].processprempted='N';
}
for(int k=0;k<=50;k++)
{
modifiedprocessqueue[k].processnumber=0;
```

```
modifiedprocessqueue[k].processtimestamp=0;
modifiedprocessqueue[k].processpriority=0;
modifiedprocessqueue[k].processwaitingtime=0;
modifiedprocessqueue[k].processprempted='N';
}
for(int i=0;i<=50;i++)
{
disp[i].processnumber=0;
disp[i].processtimestamp=0;
disp[i].processpriority=0;
disp[i].processwaitingtime=0;
disp[i].processprempted='N';
}
for(int j=0;j<=30;j++)
{
dpr[j].processor_num=0;
dpr[j].processor_efficiency=0;
dpr[j].processorwaitingtime=0;
}
}

// function to read processor information
void scheduling::read_pr_info()
{
ifstream fin_fle;

fin_fle.open("C:/Anu/prgs/sch/QT-prg/backup/gui/gui/pr_fname.txt",ios::in);
if(fin_fle.fail())
{
fstream out("errors.txt",ios::app);
out << "Error: cannot open processor data file name" <<endl;
out.close();
fin_fle.close();
exit(0);
}
fin_fle>>pr_fname;
fin_fle.close();

// Opening pdata.txt file in read mode

if((pptr=fopen(pr_fname.c_str(),"r"))==NULL)
{
fstream out("errors.txt",ios::out);
out << "Error: cannot read input file" <<endl;
out.close();
exit(1);
}
//
FILE * stream = fopen(pr_fname.c_str(),"r");

ifstream fin;
int eff=0;
fin.open(pr_fname.c_str(),ios::in);
if (fin.fail())
{
fstream out("errors.txt",ios::app);
out << "Error: cannot open processorinfo file " <<endl;
out.close();

exit(0);
```

```
}

int i=1;
while(!fin.eof())
{
fin>>dpr[i].processor_efficiency;
npr++;
dpr[i].processor_num=i;
dpr[i].processorwaitingtime=0;
i++;
}
fin.close();

for(int i=1;i<=npr;i++)
{
dpr[i].processor_efficiency=dpr[i].processor_efficiency/10;
}
for(int i=1;i<=npr;i++)
{
if(dpr[i].processor_efficiency==0)
eff++;
}
if(eff==npr)
{
exit(0);
}
for(int i=1;i<=npr;i++)
{
nprct[i]=1;
}
for(int i=1;i<=npr;i++)
{
nrq1[i]=1;
}
}

//function to read process information
void scheduling::read_process_info()
{
int i=1;
npr=0;
npwq=0;
np_tot=0;

ifstream fin;

fin.open("C:/Anu/prgs/sch/QT-prg/backup/gui/gui/fname.txt",ios::in);
if(fin.fail())
{
fstream out("errors.txt",ios::app);
out << "Error: cannot read process data file name" <<endl;
out.close();
fin.close();
exit(0);
}
fin>>fname;
fin.close();

/* Opening pdata.txt file in read mode */
```

```
if((pptr=fopen(fname.c_str(),"r"))==NULL)
{
fstream out("errors.txt",ios::out);
out << "Error: cannot open input file" <<endl;
out.close();
exit(1);
}

FILE * stream = fopen(fname.c_str(),"r");
fseek( stream, 0L, SEEK_END );

long endPos = ftell( stream );
if(endPos==0)
{
fstream out("errors.txt",ios::out);
out << "Error: empty input file " <<endl;
out.close();
fclose(pptr);
exit(1);
}
fclose( stream );

char seqNme[50];
char tmp[50];
while(!feof(pptr))
{
/* Reading the values in the file */

tmp[0]='a';
fscanf(pptr,"%s",&tmp);
if(isalpha(tmp[0]))
{
strcpy(seqNme,tmp);
}
else
{
waitingqueue[i].sequenceName = seqNme;
waitingqueue[i].preference=atoi(tmp);
fscanf(pptr,"%d",&waitingqueue[i].processtimestamp);
fscanf(pptr,"%d",&waitingqueue[i].processpriority);
fscanf(pptr,"%d",&waitingqueue[i].packetcnt);
if(waitingqueue[i].processpriority==0)
{
waitingqueue[i].processpriority=1;
}
/* Checking for the valid priority(Priority should be either 1 or 2 or 3) */
if((waitingqueue[i].processpriority<1)||(waitingqueue[i].processpriority>3))
{
fstream out("errors.txt",ios::app);
out << "Error: wrong priority value in input file" <<endl;
out.close();
exit(0);
}
waitingqueue[i].processnumber=i;
waitingqueue[i].processwaitingtime=0;
waitingqueue[i].processprempted='N';
i++;
npwq++;
}
}
```

```
np_tot=npwq;

fclose(pptr);

}
/* funtion to transfer the processes from waiting queue to the ready queue according to the
priority */

void scheduling::wq_to_rq()
{
int prlwt=0;
int npr_lwt=1;
int k=1;
for(int priority=1;priority<=3;priority++)
{
int flg=0;
for(int k=1;k<=npr;k++)
if(nrq1[k]<=dpr[k].processor_efficiency)
flg=1;
if(flg==1)
{
for(int i=1;i<=npwq;i++)
{
int f=0;

for(int k=1;k<=npr;k++)
{
if((nrq1[k]<=dpr[k].processor_efficiency)&&(f==0))
{
prlwt=dpr[k].processorwaitingtime;
npr_lwt=k;
f=1;
}
}
// Finding the processor with lowest waiting time
for(int j=1;j<=npr;j++)
{
if((prlwt>dpr[j].processorwaitingtime)&&(nrq1[j]<=dpr[j].processor_efficiency))
{
prlwt=dpr[j].processorwaitingtime;
npr_lwt=j;
}
}
if(waitingqueue[i].processpriority==priority)
{
for(int j=1;j<=npr;j++)
{
if(npr_lwt==j)
{
if((waitingqueue[i].processpriority==priority)&& (nrq1[j]<=dpr[j].processor_efficiency))
{
readyqueue[j][nrq1[j]].processnumber=waitingqueue[i].processnumber;
readyqueue[j][nrq1[j]].processtimestamp=waitingqueue[i].processtimestamp;
readyqueue[j][nrq1[j]].processpriority=waitingqueue[i].processpriority;
readyqueue[j][nrq1[j]].processwaitingtime=waitingqueue[i].processwaitingtime;
readyqueue[j][nrq1[j]].processprempted=waitingqueue[i].processprempted;

dpr[j].processorwaitingtime+=readyqueue[j][nrq1[j]].processtimestamp;

nrq1[j]++;
```

```
for(k=i;k<=npwq;k++)
{
waitingqueue[k].processnumber=waitingqueue[k+1].processnumber;
waitingqueue[k].processtimestamp=waitingqueue[k+1].processtimestamp;
waitingqueue[k].processpriority=waitingqueue[k+1].processpriority;
waitingqueue[k].processwaitingtime=waitingqueue[k+1].processwaitingtime;
waitingqueue[k].processprempted=waitingqueue[k+1].processprempted;
}
waitingqueue[k].processnumber=0;
waitingqueue[k].processtimestamp=0;
waitingqueue[k].processpriority=0;
waitingqueue[k].processwaitingtime=0;
waitingqueue[k].processprempted='N';
npwq--;
i--;

}
}
}
}
}
}
}
}

/* function to swap the higher priority processes in the waiting queue with the lower
priority ones in the ready queue */

void scheduling::swap_acc_to_priority()
{
for(int i=1;i<=npwq;i++)
{
for(int j=1;j<=npr;j++)
{
for(int i1=1;i1<=nrq1[j];i1++)
{
if(readyqueue[j][i1].processpriority>waitingqueue[i].processpriority)
{
dpr[j].processorwaitingtime-=readyqueue[j][i1].processtimestamp;

tmpar.processnumber=waitingqueue[i].processnumber;
tmpar.processtimestamp=waitingqueue[i].processtimestamp;
tmpar.processpriority=waitingqueue[i].processpriority;
tmpar.processwaitingtime=waitingqueue[i].processwaitingtime;
tmpar.processprempted=waitingqueue[i].processprempted;

waitingqueue[i].processnumber=readyqueue[j][i1].processnumber;
waitingqueue[i].processtimestamp=readyqueue[j][i1].processtimestamp;
waitingqueue[i].processpriority=readyqueue[j][i1].processpriority;
waitingqueue[i].processwaitingtime=readyqueue[j][i1].processwaitingtime;
waitingqueue[i].processprempted=readyqueue[j][i1].processprempted;


readyqueue[j][i1].processnumber=tmpar.processnumber;
readyqueue[j][i1].processtimestamp=tmpar.processtimestamp;
readyqueue[j][i1].processpriority=tmpar.processpriority;
readyqueue[j][i1].processwaitingtime=tmpar.processwaitingtime;
readyqueue[j][i1].processprempted=tmpar.processprempted;
```

```
dpr[j].processorwaitingtime+=readyqueue[j][i1].processtimestamp;
}
}
}
}
}

/* Functionality Segment.. Scheduling the processes */
int scheduling::pr_execution(char status)
{
int count=0;
int npnew=0;
int ck=0;
int i1=1;
int j1=1;
int k=1;

for(int k=1;k<=npr;k++)
nprct[k]=1;

/* Function call to load the processes in to the Ready queue */
wq_to_rq();

do
{
for(int j=1;j<=10;j++)
{
for(int i=1;i<=npr;i++)
{
if(nprct[i]>=nrq1[i])
nprct[i]=1;

if((readyqueue[i][nprct[i]].processtimestamp!=0)&& (nprct[i]<=nrq1[i]))
{
/* Incrementing the process waiting time in the ready queue(In this case Quantum is 1) */
for(int l=1;l<=nrq1[i];l++)
{
if(readyqueue[i][l].processtimestamp>0)
{
readyqueue[i][l].processwaitingtime+=100;
}
}
/* Reducing the timestamp(In this case Quantum is 1) */
readyqueue[i][nprct[i]].processtimestamp-=100;

/* Decrementing the total processors waiting time(In this case Quantum is 1) */
dpr[i].processorwaitingtime--;
if(i==1)
{
for(int l=1;l<=npwq;l++)
{
/* Incrementing the process waiting time in the waiting queue(In this case Quantum is 1) */

if(waitingqueue[l].processtimestamp>0)
{
waitingqueue[l].processwaitingtime+=100;
}

/* Changing the priority of the processes in the waiting queue, if the process is waiting for longer
time in the waiting queue*/
```

```
if((waitingqueue[l].processwaitingtime>2000)&&(waitingqueue[l].processpriority==3))
{
waitingqueue[l].processpriority=2;
/* Swaping the higher priority processes in the waiting queue with the lower priority ones in the
ready queue*/
swap_acc_to_priority();
}
if((waitingqueue[l].processwaitingtime>4000)&&(waitingqueue[l].processpriority==2))
{
waitingqueue[l].processpriority=1;
/* Swaping the higher priority processes in the waiting queue with the lower priority ones in the
ready queue*/
swap_acc_to_priority();
}
}
}
/* Changing the priority of the processes in the Ready queue, if the process is waiting for longer
time in the waiting queue*/
if((readyqueue[i][nprct[i]].processwaitingtime>2000)&&(readyqueue[i][nprct[i]].processpriority==3
))
{
readyqueue[i][nprct[i]].processpriority=2;
}
if((readyqueue[i][nprct[i]].processwaitingtime>4000)&&(readyqueue[i][nprct[i]].processpriority==2
))
{
readyqueue[i][nprct[i]].processpriority=1;
}
/* Preempting the processes, if its waiting time is too longer*/
if((readyqueue[i][nprct[i]].processwaitingtime>100000)&&(readyqueue[i][nprct[i]].processnumber
!=0))
{
disp[i1].processnumber=readyqueue[i][nprct[i]].processnumber;
disp[i1].processtimestamp=readyqueue[i][nprct[i]].processtimestamp;
disp[i1].processpriority=readyqueue[i][nprct[i]].processpriority;
disp[i1].processwaitingtime=readyqueue[i][nprct[i]].processwaitingtime;
disp[i1].processprempted='Y';


for(k=nprct[i];k<=nrq1[i];k++)
{
readyqueue[i][k].processnumber=readyqueue[i][k+1].processnumber;
readyqueue[i][k].processtimestamp=readyqueue[i][k+1].processtimestamp;
readyqueue[i][k].processpriority=readyqueue[i][k+1].processpriority;
readyqueue[i][k].processwaitingtime=readyqueue[i][k+1].processwaitingtime;
readyqueue[i][k].processprempted=readyqueue[i][k+1].processprempted;
}
readyqueue[i][k].processnumber=0;
readyqueue[i][k].processtimestamp=0;
readyqueue[i][k].processpriority=0;
readyqueue[i][k].processwaitingtime=0;
readyqueue[i][k].processprempted='N';
nrq1[i]--;

fstream out("errors.txt",ios::app);
out << "Process "<< disp[i1].processnumber <<" Preempted." <<endl;
out.close();

i1++;
}
```

```
}
count=0;
if(readyqueue[i][nprct[i]].processtimestamp==0)
{
for(int k=1;k<=nrq1[i];k++)
{
if(readyqueue[i][nprct[i]].processnumber==disp[k].processnumber)
count++;
}
/* Moving the completed processes to the display array */
if((count==0)&&(readyqueue[i][nprct[i]].processnumber!=0))
{
disp[i1].processnumber=readyqueue[i][nprct[i]].processnumber;
disp[i1].processtimestamp=readyqueue[i][nprct[i]].processtimestamp;
disp[i1].processpriority=readyqueue[i][nprct[i]].processpriority;
disp[i1].processwaitingtime=readyqueue[i][nprct[i]].processwaitingtime;
disp[i1].processprempted=readyqueue[i][nprct[i]].processprempted;

i1++;
int k=1;
for(k=nprct[i];k<=nrq1[i];k++)
{
readyqueue[i][k].processnumber=readyqueue[i][k+1].processnumber;
readyqueue[i][k].processtimestamp=readyqueue[i][k+1].processtimestamp;
readyqueue[i][k].processpriority=readyqueue[i][k+1].processpriority;
readyqueue[i][k].processwaitingtime=readyqueue[i][k+1].processwaitingtime;
readyqueue[i][k].processprempted=readyqueue[i][k+1].processprempted;
}
readyqueue[i][k].processnumber=0;
readyqueue[i][k].processtimestamp=0;
readyqueue[i][k].processpriority=0;
readyqueue[i][k].processwaitingtime=0;
readyqueue[i][k].processprempted='N';
nrq1[i]--;
}
nprct[i]--;
}
nprct[i]++;
} // end of i while

/* In place of completed processes, loading the waiting processes in to ready queue */
wq_to_rq();
/* Checking for the newly arrived processes */
/* Opening pdata.txt file in append mode */

if((pptr=fopen(fname.c_str(),"a+"))==NULL)
{
fstream out("errors.txt",ios::out);
out << "Error: cannot open input file"  <<endl;
out.close();

exit(1);
}

npnew=1;

char seqNme[50];
char tmp[50];

while(!feof(pptr))
```

```
{
tmp[0]='a';
fscanf(pptr,"%s",&tmp);
if(isalpha(tmp[0]))
{
strcpy(seqNme,tmp);
}
else
{
dtemp[npnew].sequenceName = seqNme;
dtemp[npnew].preference=atoi(tmp);
fscanf(pptr,"%d",&dtemp[npnew].processtimestamp);
fscanf(pptr,"%d",&dtemp[npnew].processpriority);
fscanf(pptr,"%d",&dtemp[npnew].packetcnt);
if(dtemp[npnew].processpriority==0)
{
dtemp[npnew].processpriority=1;
}
/* Checking for the valid priority(Priority should be either 1 or 2 or 3) */
if((dtemp[npnew].processpriority<1)||(dtemp[npnew].processpriority>3))
{
fstream out("errors.txt",ios::app);
out << "Error: wrong priority value in input file##WQ" <<endl;
out.close();
exit(0);
}
dtemp[npnew].processnumber=npnew;
dtemp[npnew].processwaitingtime=0;
dtemp[npnew].processprempted='N';

npnew++;

}
}
npnew=npnew-1;
/* Closing the file pdata.txt */
fclose(pptr);

if(npnew>np_tot)
{
/* Moving the new processes into the waiting queue*/
j1=npwq+1;
for(int k=np_tot+1;k<=npnew;k++)
{
waitingqueue[j1].processnumber=k;
waitingqueue[j1].processtimestamp=dtemp[k].processtimestamp;
waitingqueue[j1].processpriority=dtemp[k].processpriority;
waitingqueue[j1].processwaitingtime=0;
waitingqueue[j1].processprempted='N';

dtemp[k].processnumber=0;
dtemp[k].processtimestamp=0;
dtemp[k].processpriority=0;
dtemp[k].processwaitingtime=0;
dtemp[k].processprempted='N';
npwq++;
j1++;
}
np_tot=npnew;
/* Moving the new processes into the ready queue*/
```

```
wq_to_rq();
swap_acc_to_priority();
}
}// end of j while
/* Printing all the Executed processes with the total time taken for execution of each process*/

if(i1==np_tot+1)
{
if(status=='Y')
{
status='X';
Processdata process_present[50];

read_process_info();
read_pr_info();

FILE *pptr;
if((pptr=fopen(fname.c_str(),"r"))==NULL)
{
fstream out("errors.txt",ios::out);
out << "Error: cannot open the input file"  <<endl;
out.close();
exit(0);
}

int pcnt=1;
while(!feof(pptr))
{
/* Reading the values in the file */

fscanf(pptr,"%d",&process_present[pcnt].processtimestamp);
fscanf(pptr,"%d",&process_present[pcnt].processpriority);

process_present[pcnt].processnumber=pcnt;
process_present[pcnt].processwaitingtime=0;
process_present[pcnt].processprempted='N';

pcnt++;
}
pcnt--;
fclose(pptr);
for(int i=1;i<=np_tot;i++)
{

for(int j=1;j<=pcnt;j++)
{
if(process_present[j].processnumber==disp[i].processnumber)
{
process_present[j].processtimestamp=0;
process_present[j].processpriority=0;
process_present[j].processnumber=0;
process_present[j].processwaitingtime=0;
process_present[j].processprempted='N';


ofstream outdata; // outdata is like cin
outdata.open(fname.c_str()); // opens the file
if( !outdata )
{ // file couldn't be opened
fstream out("errors.txt",ios::out);
```

```
out << "Error: cannot open input file:"  <<endl;
out.close();

exit(0);
}
for (int i=1; i<=pcnt; ++i)
{
if(process_present[i].processnumber!=0)
{
outdata << process_present[i].processtimestamp<< endl;
outdata << process_present[i].processpriority;
if(i<pcnt)

{
outdata<< endl;
}
}
}
outdata.close();
}
}
}
return disp[np_tot].processwaitingtime;
}
if(status=='E')
{
return disp[np_tot].processwaitingtime;
}
}
}while(i1<=np_tot); //end of do while loop
}

// Function to print the calculated result in the result.txt file
void scheduling::print()
{
sc_pvector<int> pr_eff;
int result=0;

char cmd;
int temp;
ifstream command;
command.open("C:/Anu/prgs/sch/QT-prg/backup/gui/gui/command.txt",ios::in);
assert (!command.fail( ));

command>>cmd;
command.close();

if(cmd=='E')
{
initialise();
read_process_info();
read_pr_info();
result=pr_execution('E');


ofstream fout;
fout.open("C:/Anu/prgs/sch/QT-prg/backup/gui/gui/result.txt",ios::out);
assert (!fout.fail( ));

fout<<result;
```

```
fout.close();
}
if(cmd=='Y')
{
initialise();
read_process_info();
read_pr_info();
result=pr_execution('Y');

ofstream fout;
fout.open("C:/Anu/prgs/sch/QT-prg/backup/gui/gui/result.txt",ios::out);
assert (!fout.fail( ));

fout<<result;

fout.close();

}
}
```

gui.h

```
//*********************************************************************
//header file for GUI input-output data
//*********************************************************************

#ifndef GUI_H
#define GUI_H

#include <QtGui/QMainWindow>
#include <QtGui/QApplication>
#include<QProcess>
#include "XMLFileParser.h"
#include "ui_gui.h"

class MyClass : public QMainWindow
{
Q_OBJECT

public:
MyClass(QWidget *parent = 0, Qt::WFlags flags = 0);
~MyClass();
void command(char);
void processorinfo();
QString read_est();
void multhread();
void display_errors();

public slots:
void quit();
void exefun();
void estfun();
void reset();
void reset_process();
void pr_display();
void update_pr_info();
void pr_filename();
void XML_filename();

private:
```

```
Ui::MyClassClass ui;
};

#endif // GUI_H
```

gui.cpp

```
//********************************************************************
// Functions for GUI accessing and printing data
//********************************************************************

#include "gui.h"
#include <QFile.h>
#include<QString.h>
#include <QTextStream>
#include<QVector>

// function to connect the buttons to resp functionalities
MyClass::MyClass(QWidget *parent, Qt::WFlags flags)
: QMainWindow(parent, flags)
{
ui.setupUi(this);

connect(ui.pushButton_1,SIGNAL(clicked()),this,SLOT(estfun()));
connect(ui.pushButton_2,SIGNAL(clicked()),this,SLOT(exefun()));

connect(ui.pushButton_4,SIGNAL(clicked()),this,SLOT(reset()));
connect(ui.pushButton_3,SIGNAL(clicked()),this,SLOT(quit()));
connect(ui.pushButton_6,SIGNAL(clicked()),this,SLOT(XML_filename()));

connect(ui.pushButton_5,SIGNAL(clicked()),this,SLOT(pr_filename()));
connect(ui.display_button,SIGNAL(clicked()),this,SLOT(pr_display()));
connect(ui.update_pr,SIGNAL(clicked()),this,SLOT(update_pr_info()));
}

void MyClass::XML_filename()
{
QString fname=ui.lineEdit_3->displayText();

JCParsingQt::XMLFileParser fileParser;
fileParser.readFile( fname, QString( "data.txt" ) );


QString OutputFname="data.txt";

QString path="C:/Anu/prgs/sch/QT-prg/backup/gui/gui/";
QString completefilepath=path+OutputFname;

QFile file("fname.txt");

file.open(QIODevice::WriteOnly | QIODevice::Text);
QTextStream out( &file );

out<<completefilepath;

reset_process();

file.close();

}
```

```
// function to print the given file name in fname.txt file
void MyClass::reset_process()
{
ui.lineEdit_4->setText(0);
ui.textEdit->clear();
ui.pr_text->clear();

QFile file("command.txt");
file.open(QIODevice::WriteOnly | QIODevice::Text);
file.close();

QFile file3("result.txt");
file3.open(QIODevice::WriteOnly | QIODevice::Text);
file3.close();

QFile file4("errors.txt");
file4.open(QIODevice::WriteOnly | QIODevice::Text);
file4.close();
}
// function to print the given file name in fname.txt file
void MyClass::pr_filename()
{
QString pr_fname=ui.lineEdit_2->displayText();

QString path="C:/Anu/prgs/sch/QT-prg/backup/gui/gui/";
QString completefilepath=path+pr_fname;

QFile file("pr_fname.txt");

file.open(QIODevice::WriteOnly | QIODevice::Text);
QTextStream out( &file );

out<<completefilepath;

reset_process();

file.close();
}
// function to print the given command in command.txt
void MyClass::command(char ch)
{
QFile file("command.txt");

file.open(QIODevice::WriteOnly | QIODevice::Text);
QTextStream out( &file );

out<<ch<<endl;

file.close();
}
// function to display processor file contents
void MyClass::pr_display()
{
ui.pr_text->clear();

QVector<int> pr_eff;


QString pr_fname=ui.lineEdit_2->displayText();
```

```
QFile file1(pr_fname);

if ( file1.open( QIODevice::ReadOnly ) )
{
QTextStream dta(&file1);

while(!dta.atEnd())
{
ui.pr_text->append( dta.readLine());
}

}
file1.close();
}

// function to display processor file contents
void MyClass::update_pr_info()
{
QVector<int> pr_eff;
int data;

QString update_data=ui.pr_text->toPlainText();

QString pr_fname=ui.lineEdit_2->displayText();
QFile file(pr_fname);
file.open(QIODevice::WriteOnly | QIODevice::Text);

QTextStream out( &file );
out << update_data;
file.close();
}

// function to extract processor efficiency values in to a vector
void MyClass::processorinfo()
{
QVector<int> pr_eff;
QString pr_fname=ui.lineEdit_2->displayText();
int data;
QFile file(pr_fname);
file.open(QIODevice::ReadOnly | QIODevice::Text);

if ( file.open( QIODevice::ReadOnly ) )
{
QTextStream in(&file);

while(!in.atEnd())
{
in>>data;
pr_eff.push_back(data);
}
}
file.close();
}

//function to read the result value from the result.txt file
QString MyClass::read_est()
{
QString result;
QFile file("result.txt");
```

```
file.open(QIODevice::ReadOnly | QIODevice::Text);
QTextStream in(&file);

in>>result;
file.close();
return result;
}

//function to display errors from the errors.txt file
void MyClass::display_errors()
{

QString err;
QFile file( "errors.txt" ); // Read the text from a file
if ( file.open( QIODevice::ReadOnly ) )
{
QTextStream stream( &file );

while(!stream.atEnd())
{
ui.textEdit->append(stream.readLine());
}
}
file.close();
}
// funtion to reset all the values
void MyClass::reset()
{
ui.lineEdit_4->setText(0);
ui.lineEdit_3->clear();
ui.lineEdit_2->clear();
ui.textEdit->clear();
ui.pr_text->clear();

QFile file("command.txt");
file.open(QIODevice::WriteOnly | QIODevice::Text);
file.close();

QFile file1("fname.txt");
file1.open(QIODevice::WriteOnly | QIODevice::Text);
file1.close();

QFile file5("pr_fname.txt");
file5.open(QIODevice::WriteOnly | QIODevice::Text);
file5.close();

QFile file3("result.txt");
file3.open(QIODevice::WriteOnly | QIODevice::Text);
file3.close();

QFile file4("errors.txt");
file4.open(QIODevice::WriteOnly | QIODevice::Text);
file4.close();

}

//function to reset values before quiting
void MyClass::quit()
{
```

```
reset();
}
// funtion to invoke functionality exe file
void MyClass::multhread()
{
QApplication::setOverrideCursor( Qt::WaitCursor );

QString program = "C:/Anu/prgs/sch/QT-
prg/backup/Simuation_scheduling/Simuation_scheduling/Debug/ Simuation_scheduling.exe";
QStringList arguments;

QProcess* process = new QProcess( this );
process->start( program, arguments );

// If this returns false, something went wrong
bool started = process->waitForStarted( 2000 );

unsigned waits = 0;

// Wait one second at a time for the process' end
while( !process->waitForFinished( 1000 ) )
{
++waits;

// If process won't end within one minute, terminate it
if( 60 < waits )
{
process->terminate( );
break;
}
}

// When the new process ends, execution is here

delete process;
process = NULL;

QApplication::restoreOverrideCursor( );

}

// function to invoke necessary functionality to give an estimate time for scheduling
void MyClass::estfun()
{
QFile file4("errors.txt");
file4.open(QIODevice::WriteOnly | QIODevice::Text);
file4.close();
ui.textEdit->clear();

QFile file("result.txt");
file.open(QIODevice::WriteOnly | QIODevice::Text);
file.close();
processorinfo();
command('E');
multhread();

QString result = read_est();
ui.lineEdit_4->setText(result);

display_errors();
```

```
}

// function to invoke necessary functionality to execute the scheduling
void MyClass::exefun()
{
QFile file4("errors.txt");
file4.open(QIODevice::WriteOnly | QIODevice::Text);
file4.close();
ui.textEdit->clear();

QFile file("result.txt");

file.open(QIODevice::WriteOnly | QIODevice::Text);

file.close();

processorinfo();
command('Y');
multhread();
QString result = read_est();
ui.lineEdit_4->setText(result);

display_errors();
}
```